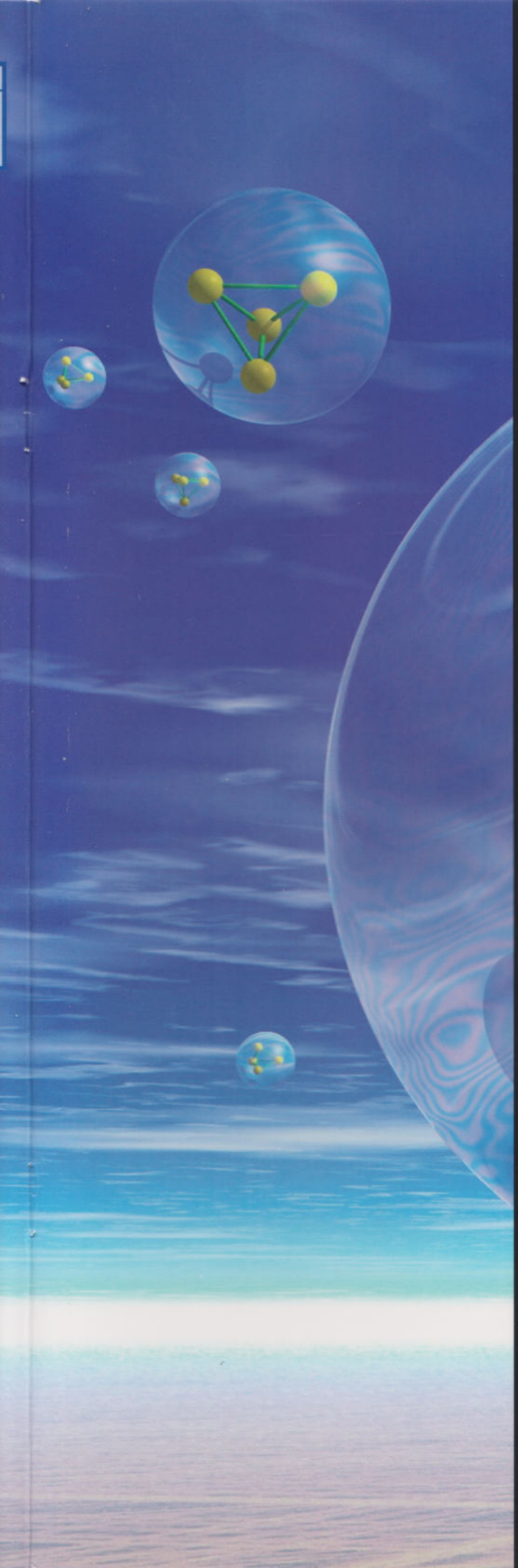
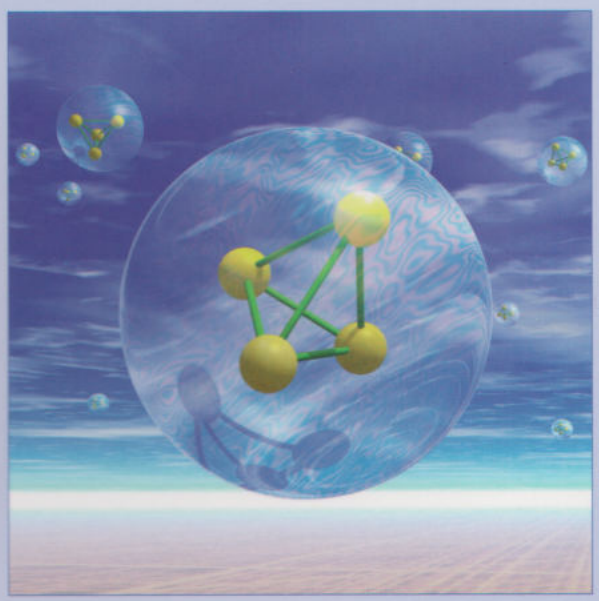


The Open University

M255 Unit 12
UNDERGRADUATE COMPUTING

Object-oriented programming with Java



Streams, files and persistent objects

Unit 12

M255 Unit 12
UNDERGRADUATE COMPUTING

Object-oriented programming with Java

Streams, files and persistent objects

Unit 12

This publication forms part of an Open University course M255 *Object-oriented programming with Java.* Details of this and other Open University courses can be obtained from the Student Registration and Enquiry Service, The Open University, PO Box 197, Milton Keynes, MK7 6BJ, United Kingdom: tel. +44 (0)870 333 4340, email general-enquiries@open.ac.uk

Alternatively, you may visit the Open University website at http://www.open.ac.uk where you can learn more about the wide range of courses and packs offered at all levels by The Open University.

To purchase a selection of Open University course materials visit http://www.ouw.co.uk, or contact Open University Worldwide, Michael Young Building, Walton Hall, Milton Keynes, MK7 6AA, United Kingdom for a brochure: tel. +44 (0)1908 858785; fax +44 (0)1908 858787; email ouwenq@open.ac.uk

The Open University
Walton Hall
Milton Keynes
MK7 6AA

First published 2006. Second edition 2008.

Edited and designed by The Open University.

Typeset by The Open University.

Printed and bound in the United Kingdom by The Charlesworth Group, Wakefield.

ISBN 978 0 7492 6793 3

2.1

The paper used in this publication contains pulp sourced from forests independently certified to the Forest Stewardship Council (FSC) principles and criteria. Chain of custody certification allows the pulp from these forests to be tracked to the end use (see www.fsc.org).

CONTENTS

Introduction 5

1 Files and streams 6

1.1 File and stream classes in `java.io` 6

1.2 The `File` class 9

2 Simple writing and reading of text files 15

2.1 Exceptions, files and streams 15

2.2 Writing to a file 17

2.3 Reading from a file 21

3 Buffering and wrapping classes 24

3.1 Buffers 24

3.2 Wrapping stream classes 25

3.3 Writing to a file using a `BufferedWriter` 27

3.4 Reading from a file using a `BufferedReader` 31

4 Making objects persistent using text files 33

4.1 Writing the details of `Account` objects to file 33

4.2 Writing files that can be used to recreate objects 36

4.3 Reading a text file using the `Scanner` class 38

4.4 Summary of stream classes, messages and exceptions 46

5 Persistence through serialisation 49

5.1 The `Serializable` interface 50

5.2 Reading and writing serialised objects 51

5.3 The limitations of serialisation 55

6 Streams and networks 56

7 Summary 58

Glossary 60

Index 61

M255 COURSE TEAM

Affiliated to The Open University unless otherwise stated.

Rob Griffiths, Course Chair, Author and Academic Editor

Lindsey Court, Author

Marion Edwards, Author and Software Developer

Philip Gray, External Assessor, University of Glasgow

Simon Holland, Author

Mike Innes, Course Manager

Robin Laney, Author

Sarah Mattingly, Critical Reader

Percy Mett, Academic Editor

Barbara Segal, Author

Rita Tingle, Author

Richard Walker, Author and Critical Reader

Robin Walker, Critical Reader

Julia White, Course Manager

Ian Blackham, Editor

Phillip Howe, Compositor

John O'Dwyer, Media Project Manager

Andy Seddon, Media Project Manager

Andrew Whitehead, Graphic Artist

Thanks are due to the Desktop Publishing Unit, Faculty of Mathematics and Computing.

Introduction

This unit is about exchanging information between a Java program and an external source, thereby enabling Java programs to communicate with the wider world. It is also about making information **persistent**, so that it can be stored at one time and retrieved unchanged at a later time. For example, if you are word processing a document, you expect to be able to save it one day and open the document the next day and find it in the same state as you left it. Persistence involves saving data to files.

So far in M255 you have made information persistent using the features of both BlueJ and the OUWorkspace: you have made changes to classes and saved those changes to a file; you have created and saved new classes; and you may have saved statements in the OUWorkspace's Code Pane using the Save or Save As... options from the File menu.

What you have not been able to do in M255, so far, is to make objects persistent. For example, if you have created an account object in the OUWorkspace, you have not been able to save that account so that you can retrieve it during a subsequent session of the OUWorkspace. In this unit you will learn how to make objects persistent, so that they may be created at one time on one computer and retrieved at another time, possibly on a different computer.

In Section 1 you learn about Java's `io` (input/output) library and the file and stream classes it contains.

Section 2 starts off by revisiting exceptions, which were introduced in *Unit 8*, and explains why they are important when working with files and streams. You go on to learn about writing and reading characters to and from text files.

Section 3 introduces the concept of buffering, for improving the efficiency of writing and reading to and from files. You learn what buffers are and when you should use them.

In Section 4 you learn how to make objects persistent by saving the textual representation of their state to text files and then reading those files to recreate the objects.

Section 5 details another strategy for making objects persistent: saving them as raw bytes to a file. This is called **serialisation**.

Section 6 is a short introduction to how streams can be used to transfer information across networks.

1 Files and streams

1.1 File and stream classes in `java.io`

What is a stream? In the physical world, a stream of water flows from a source (for example, a spring) down to a sink (for example, a pond). In the programming world, we may have a data source (for example, a program generating a list of random numbers) and a data sink (a file where we want to store these numbers). A **stream** is the object we use to link them, enabling data to be transferred from the source to the sink (Figure 1).

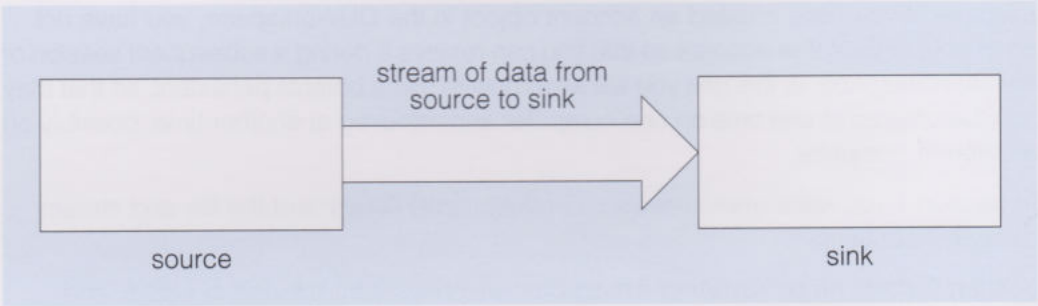


Figure 1 A data stream

In most of this unit we are looking at streams which connect a Java program and a file. When we want to save data that is currently in a Java program to a file, the source is the Java program and the sink is the file, and the Java program writes data to the stream (Figure 2).

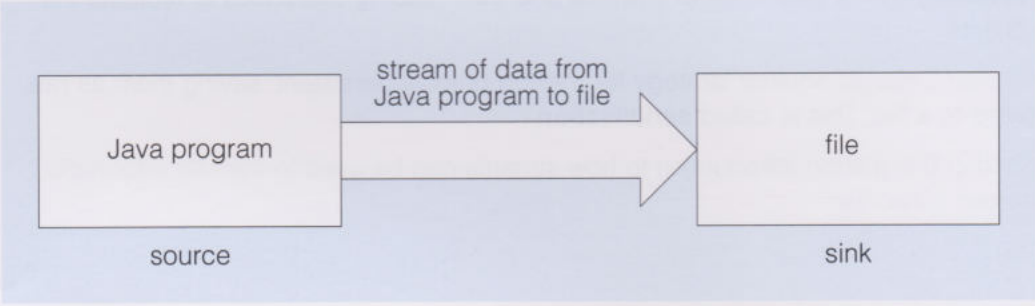


Figure 2 Saving data to a file

When we want to read data into a Java program, the source is the file and the sink is the Java program, and the Java program reads data from the stream (Figure 3).

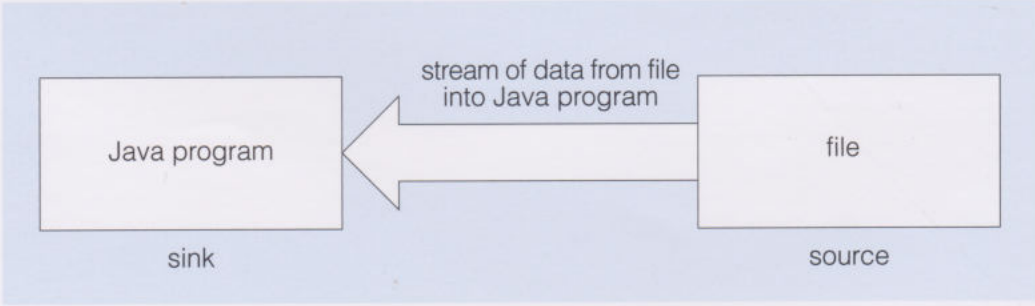


Figure 3 Reading data from a file

Reading from (or writing to) a stream is done sequentially: when you read data from a stream the first item is read, then the second, and so on until the last item is reached.

ACTIVITY 1

Java has a large number of classes to support the reading and writing of data using streams. The purpose of this activity is to give you some feel for the number of these classes and the naming conventions they use. Launch BlueJ and open the documentation for the Java Class Libraries (select the Help menu then Java Class Libraries). In the main frame, scroll down and select the package `java.io`. Scroll down to the section labelled 'Class Summary', look at the class names, and see if you can use them to identify related groups of classes. You are not expected to remember all the class names; nor are you expected to understand everything in the descriptions. You should spend no more than 10 minutes on this activity.

DISCUSSION OF ACTIVITY 1

You have probably identified groups of classes whose names end with `InputStream` or `OutputStream` and others whose names end with `Reader` or `Writer`. You will probably have (correctly) deduced that `FileReader`, `BufferedReader` etc. are subclasses of the `Reader` class. You will also have noticed that there are groups of classes whose names start with the same word, such as `FileInputStream`, `FileOutputStream`, `FileReader` and `FileWriter` (and a similar group whose names start with `Buffered`), and you have probably concluded that these classes carry out similar tasks.

The number of classes available for input and output is daunting, and in M255 we will only look at a small number of them. However, for completeness, Table 1 (overleaf) lists all the `java.io` stream classes, which fall into four main groups subclassed from `InputStream`, `OutputStream`, `Reader` and `Writer`.

The stream classes which you will be using during M255 are in bold in Table 1. To read the table: the class `InputStream` has the subclasses `ByteArrayInputStream`, `FileInputStream` etc.; the class `FilterInputStream` has the subclasses `BufferedInputStream`, `DataInputStream`, etc.

There are two groups of stream classes to read data (`InputStream` and `Reader`) and two groups of classes to write data (`OutputStream` and `Writer`). If you were reading and writing data to/from a sink/source, you would pair an `InputStream` class with an `OutputStream` class and a `Reader` class with a `Writer` class. The difference between the two groups of stream classes is in the type of data which they handle.

- Instances of subclasses of the `Reader` and `Writer` classes handle (16-bit) character streams. This means that they correctly handle textual information based on characters and strings. We will use an instance of a subclass of the `Writer` class (`FileWriter`) to write text to a file in Subsection 2.2.
- Instances of subclasses of the `InputStream` and `OutputStream` classes handle (8-bit) byte streams. They are used when we make objects persistent through serialisation (Section 5), and for writing binary data such as sounds and images.

Table 1 Java Stream classes in the `java.io` package

InputStream	ByteArrayInputStream	
	FileInputStream	
	FilterInputStream	**BufferedInputStream**
		DataInputStream
		LineNumberInputStream
		PushbackInputStream
	ObjectInputStream	
	PipedInputStream	
	SequenceInputStream	
	StringBufferInputStream	
OutputStream	ByteArrayOutputStream	
	FileOutputStream	
	FilterOutputStream	**BufferedOutputStream**
		DataOutputStream
		PrintStream
	ObjectOutputStream	
	PipedOutputStream	
Reader	**BufferedReader**	LineNumberReader
	CharArrayReader	
	FilterReader	PushbackReader
	InputStreamReader	**FileReader**
	PipedReader	
	StringReader	
Writer	**BufferedWriter**	
	CharArrayWriter	
	FilterWriter	
	OutputStreamWriter	**FileWriter**
	PipedWriter	
	PrintWriter	
	StringWriter	
RandomAccessFile		

1.2 The `File` class

If we want to read from, or write to, files on the hard disk, then we need a way to specify which file (or folder) we are interested in. The Java class that we use to do this is the `File` class. The `File` class has a misleading name: you might think that an instance of the class would represent a physical file on a hard disk, but it does not. It represents either the *name* of a particular file or the *name* of a folder (directory).

`C:\BlueJ\README.TXT` is an example of how the Windows operating system expects users and programs to specify file and folder pathnames. However, different operating systems specify file pathnames in different formats; for example, Windows uses a '\' (backslash) to separate file and folder names whereas Macintosh OSX, Unix and Linux uses a '/'. The purpose of the `File` class is to allow pathnames to be represented in an *abstract* or system-independent way and, when required, to convert them automatically into the system-dependent format needed to access a particular physical file or folder. In M255 we are only concerned with files and folders on the Windows operating system, but you need to be aware that different operating systems do things slightly differently, and because Java programs can be executed on different platforms they have to be able to access file systems in platform-specific ways.

File pathnames can be absolute or relative. A relative pathname assumes the current working directory as the starting point, whereas an absolute pathname contains all the information you need to know about the location of a file and always starts from the root directory of the disk, usually `C:` in Windows.

`C:\BlueJ\README.TXT` is an example of an absolute pathname. In M255 we are only concerned with absolute pathnames.

In a program a pathname is represented as a string. However, this presents us with a slight problem because when a backslash is encountered in a literal string it is interpreted by the compiler as the **escape character**: it indicates that the next character should be interpreted in some special way. For example, in the following string:

```
"string\twine"
```

the backslash combines with the very next character to form what is called an escape sequence – in this case the escape sequence `\t`, which represents a tab. Therefore executing the following statement in the Code Pane:

```
System.out.println("string\twine");
```

would output:

```
string wine
```

to the Display Pane. Fortunately there is a solution to this. In order to get a backslash interpreted as just a backslash in a literal string we just precede the backslash by another backslash. For example:

```
"string\\twine"
```

Now executing the following statement in the Code Pane:

```
System.out.println("string\\twine");
```

correctly displays:

```
string\twine
```

So to create a string to represent the pathname `C:\BlueJ\README.TXT` we would write:

```
String pathname = "C:\\BlueJ\\README.TXT";
```

In Java you can use a string which represents a pathname to create an instance of the class `File`, and you will see how to do this shortly.

When a string is used to specify a pathname it must be exactly right, and if you are typing the pathname it is easy to make a mistake. So in order to simplify the specification of the correct string to represent a pathname, we have provided you with the utility class `OUFileChooser`. The `OUFileChooser` class has class methods for creating pathnames, one of which displays a dialogue box, which we call a 'file chooser' dialogue box. Such a dialogue box displays the contents of a folder (a list of physical files) and allows you to select a particular file, whose pathname is then returned when you click on the OK button. By using such dialogue boxes you do not have to worry about determining the correct absolute pathname or using `\\` within the string.

ACTIVITY 2

In this activity you investigate the `OUFileChooser`. Launch BlueJ then open the Help menu and select OU Class Library. Browse the documentation for the `OUFileChooser` class (scroll down the text until you reach the Method Summary).

Depending on your computer, and particularly if it is connected to a network, it may take a few seconds for the file chooser dialogue box to appear.

1 Open an OUWorkspace without first opening a project, then enter, select and execute the following statement:

```
String pathname = OUFileChooser.getFilename();
```

When the dialogue box opens, the contents of which folder are displayed?

Select a file listed in the dialogue box and accept it using the OK button. Look at the value of `pathname` either by inspecting `pathname` or by executing the following statement:

```
System.out.println(pathname).
```

2 Again, execute the statement

```
String pathname = OUFileChooser.getFilename();
```

but this time select the Cancel button. Check the value of `pathname`.

3 Close the OUWorkspace and then open the project Unit12_Project_1. You will notice that this project has no classes – this is intentional; the folder of this project is simply being used to read and write files from the OUWorkspace. Reopen the OUWorkspace and execute the following statement:

```
String pathname = OUFileChooser.getFilename();
```

Note the folder on which the file chooser dialogue box is focused, select a file and then click on the OK button. Check the value of `pathname`.

4 Execute the statement

```
String newPathname = OUFileChooser.getFilename();
```

but this time do not select an existing file. Instead type the name of a file which does not exist, such as `foo.txt`. Check the value of `newPathname`. You might like to do this twice, the first time entering the name of a file that exists in the folder, and the second time entering the name of a file that does not exist in the folder.

5 Execute the statements

```
String pathname1 = OUFileChooser.getFilename("README.TXT");
String pathname2 = OUFileChooser.getFilename("newFile.txt");
```

and then inspect `pathname1` and `pathname2`.

DISCUSSION OF ACTIVITY 2

From the documentation, you can see that the `OUFileChooser` class extends the `JFileChooser` class by providing three additional class (static) methods. The methods you will use to get pathnames have the signatures `getFilename()` and `getFilename(String)`.

1 When you execute the statement, the following dialogue box is shown (Figure 4). The dialogue box will be focused on the BlueJ folder, which is the location where you chose to install BlueJ. If you installed BlueJ in the default location, and you selected the file `README.TXT`, `pathname` will reference the string `"C:\BlueJ\README.TXT"`.

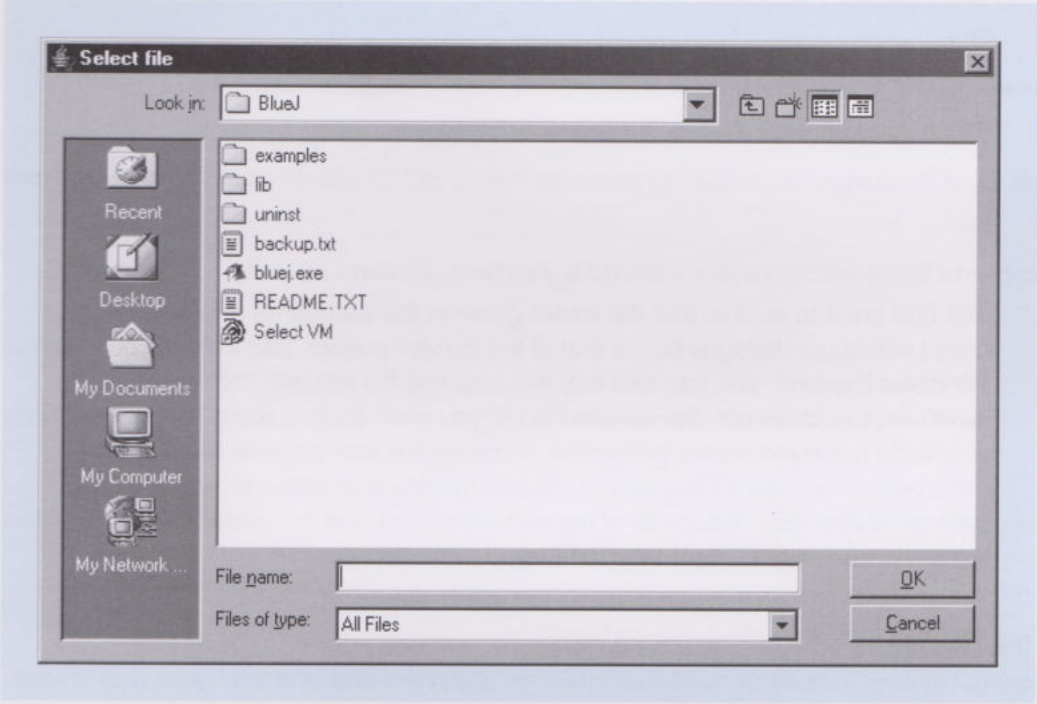


Figure 4 A file chooser dialogue box that has been opened in the OUWorkspace when a BlueJ project has not been opened

2 When you close the dialogue box using the Cancel button, `pathname` will have the value `null`, indicating that a pathname has not been returned.

3 The dialogue box will be focused on the Unit12_Project_1 folder, and if you installed the M255 software in the default location and again you selected the file `README.TXT`, `pathname` will reference either:

`"C:\Documents and Settings\<username>\My Documents\M255\M255Projects\Unit12_Project_1\README.TXT"`

or:

`"C:\My Documents\M255\M255Projects\Unit12_Project_1\README.TXT"`

depending on your operating system.

4 On this occasion `pathname` will start with either:

`"C:\Documents and Settings\<username>\My Documents\M255\M255Projects\Unit12_Project_1\"`

or:

`"C:\My Documents\M255\M255Projects\Unit12_Project_1"`

and the final part of the pathname string will be the file name you entered.

For example, if you typed `myfile.txt` in the input box, the full pathname string will be either:

```
"C:\Documents and Settings\<username>\My Documents\M255\M255Projects\Unit12_Project_1\myfile.txt"
```

or:

```
"C:\My Documents\M255\M255Projects\Unit12_Project_1\myfile.txt"
```

A pathname string will be returned regardless of whether or not the corresponding file exists (which is what you need if you want to save data to a new file).

5 In this case a dialogue box is not displayed. The variable `pathname1` now references the string:

```
"C:\Documents and Settings\<username>\My Documents\M255\M255Projects\Unit12_Project_1\README.TXT"
```

or:

```
"C:\My Documents\M255\M255Projects\Unit12_Project_1\README.TXT "
```

and `pathname2` references the following string:

```
"C:\Documents and Settings\<username>\My Documents\M255\M255Projects\Unit12_Project_1\newFile.txt"
```

or:

```
"C:\My Documents\M255\M255Projects\Unit12_Project_1\newFile.txt"
```

The first point to note is that the folder given in the pathname returned by the `OUFileChooser` dialogue box is that of the current project. Secondly, if you check in Windows Explorer, you can see that the physical file `README.TXT` exists, but `newFile.txt` does not. So, as with Part 4, you can obtain a string that corresponds to a valid but non-existent pathname. Invoking the method with the signature `OUFileChooser.getFilename(String)` provides a shortcut to selecting the pathname of a file in the folder of the current project, but remember that no check is made as to whether or not the file exists.

The previous activity has shown how the `OUFileChooser` class provides two `getFilename()` methods (one that takes an argument and one that does not). These allow us to create strings which correspond either to the pathnames of existing physical files or to the pathnames of files which we might want to create. You must remember that both the `getFilename()` methods only return strings, and a further step is needed before we have something that can actually refer to a file.

The next step is to create an instance of the `File` class, and we do this by using the string returned by a `getFilename()` method as the argument to the `File` class constructor:

```
String pathname = OUFileChooser.getFilename();
File aFile = new File(pathname);
```

After the above code executes, the variable `aFile` references a new `File` object, which may or may not correspond in some way to an existing physical file on the hard disk. Whether it does or not will depend on whether you used a `getFilename()` method to return the pathname of an existing physical file or to return a pathname that you intend to associate with a new, as yet uncreated, physical file.

It is important to note that a `File` object cannot be written to, or read from. That is not its purpose: it does not represent the contents of some physical file. However, a `File` object does hold important information, for example: the pathname; whether the pathname identifies a file or a folder (directory); whether a physical file exists at that pathname; and, if a physical file exists, whether it can be written to.

Therefore, the protocol of `File` objects includes the following messages:

- `exists()` – returns `true` if the file or directory denoted by the pathname exists; `false` otherwise.
- `isFile()` – returns `true` if the file denoted by the pathname exists and is a file; `false` otherwise.
- `isDirectory()` – returns `true` if the file denoted by the pathname exists and is a directory; `false` otherwise.
- `canWrite()` – returns `true` if the file denoted by the pathname exists and the current program is allowed to write to that file; `false` otherwise.

ACTIVITY 3

If it is not already open, launch BlueJ and open the project Unit12_Project_1. Write a sequence of statements in the OUWorkspace to do the following.

1 Get the pathname of a file called `README.TXT` by using a file chooser dialogue box and typing the file name `README.TXT` into the input box. Note that the `OUFileChooser` dialogue does *not* automatically add an extension to a pathname (because a `.txt` extension is not always appropriate), so you must remember to include the extension.

2 Create a `File` object using the pathname obtained in step 1.

3 Using an `if-then-else` statement, test whether the new file object is associated with an actual physical file on disk by sending it an `exists()` message. If the file does exist, use an alert dialogue box to display the message, 'A physical file exists!' If no physical file exists, use an alert dialogue box to display the message, 'No physical file exists!'

Repeat steps 1–3 using the file names `backup.txt` and `myfile.txt`.

Finally, what happens when you select and execute your complete code, and then you click the Cancel button on the file chooser dialogue box?

DISCUSSION OF ACTIVITY 3

Your code should look similar to the following.

```
String pathname = OUFileChooser.getFilename();
File aFile = new File(pathname);
if (aFile.exists())
{
   OUDialog.alert("A physical file exists!");
}
else
{
   OUDialog.alert("No physical file exists!");
}
```

If you enter `README.TXT` into the file chooser dialogue box, the alert dialogue box should display:

```
A physical file exists!
```

If you enter `backup.txt` into the file chooser dialogue box, the alert dialogue box should again display:

```
A physical file exists!
```

If you enter `myfile.txt` into the file chooser dialogue box, the alert dialogue box should display:

```
No physical file exists!
```

If you click the Cancel button of the file chooser dialogue box, the following error message is shown in the Display Pane:

```
Exception: line 2. java.lang.NullPointerException
```

This indicates that there was a problem evaluating the statement:

```
File aFile = new File(pathname);
```

If you inspect the variable `pathname`, you will find that it is `null`. This is perfectly reasonable, as you did not supply a file name to the file chooser dialogue box, and when the `File` constructor is presented with a `null` argument it fails. Exceptions are discussed in more detail in the next section.

SAQ 1

Thinking back to what you learnt in *Unit 8*, what sort of error occurs when you execute the statement

```
File aFile = new File(pathname);
```

and `pathname` is `null`?

ANSWER

The error is a runtime error (more precisely a dynamic semantic error), because whether or not an error is thrown depends on the value of `pathname` at run-time. It is also an example of an unchecked exception, specifically a `NullPointerException`. We know that is an unchecked exception because the OUWorkspace interpreter did not insist that we embed the call of the constructor within a `try–catch` statement.

In the next section we shall see that when using files there are many kinds of exceptions that may occur, and most are ones that the BlueJ Java compiler or the OUWorkspace's Java interpreter will force you to try and catch (i.e. checked exceptions).

2 Simple writing and reading of text files

In this section we shall look at simple ways of writing and reading characters to and from text files. Error detection and recovery is particularly important when working with file input and output. This is because a program does not control its external environment. For example, if a program requires input from a file, and that file does not exist, then the problem must be handled in such a way that the program does not fail, or if it does fail at least it does so gracefully. So we start this section by looking at what errors can occur and the steps you must take to handle those errors.

2.1 Exceptions, files and streams

Unit 8 introduced you to the idea of errors and the need to catch exceptions.

SAQ 2

Which statement do you use to trap an exception thrown by a method?

ANSWER.....

The `try–catch` statement is used.

Exercise 1

We have indicated that non-existence of a file is a potential problem when reading from files. Suggest some other potential problems of reading from and writing to files.

Solution.....

Possible problems include:

- trying to overwrite a file which is read-only;
- trying to write to a file when there is no space on the disk;
- trying to read from a file which has become corrupted – so though it exists, it does not contain the data expected by the program.

In *Unit 8* you learnt about **unchecked exceptions**. Unchecked exceptions should not occur in normal program use, and their occurrence usually indicates that the programmer has failed to take into account a problem which was predictable and should have been guarded against. Programming errors that can result in unchecked exceptions include: failing to test that an index is within the bounds of an array; dividing a number by zero; and, as in the final part of Activity 3, using `null` where an object was expected. In this last case, the test needed was

```
(pathname != null)
```

and only if this was `true` should the statement `File aFile = new File(pathname);` have been executed. Alternatively the statement `File aFile = new File(pathname);` could have been put within the `try` block of a `try–catch` statement, with the `catch` block taking remedial action if an exception occurred. Unchecked exceptions all inherit

from the class `RuntimeException`, and the Java compiler does not force a programmer to try and catch them.

The problems identified in Exercise 1 are all **checked exceptions**. For example, trying to read from a file which has become corrupt will throw a checked exception. We cannot know in advance whether a file has become corrupt, but we are aware that it might happen and need to check that the file has been read successfully. Checked exceptions have to be caught in a `try–catch` statement. Checked exceptions are known as such because the Java compiler checks that any code which can throw a checked exception is inside a `try–catch` statement, ensuring that if an exception occurs the `catch` block will handle the exception in some way. If you try to compile code which could result in a checked exception and you do not surround it by a `try–catch` statement you will get a compile-time error message such as:

```
unreported exception java.io.FileNotFoundException; must be caught or
declared to be thrown
```

You then need to identify what code throws the checked exception and put it in within an appropriate `try–catch` statement.

You can open the Java Class Libraries documentation from BlueJ's Help menu.

The question then becomes: how do we know if a method or constructor throws a checked exception? The answer is to look in the documentation of the Java Class Libraries.

If you look at the documentation for constructors from the classes `FileWriter` (which you will use in the next subsection) and `File`, you can see how to distinguish between checked and unchecked exceptions:

> `public FileWriter(File file) throws IOException`
>
> Constructs a `FileWriter` object given a `File` object.
>
> **Parameters:**
>
> `file` – a `File` object to write to.
>
> **Throws:**
>
> `IOException` – if the file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason

> `public File (String pathname)`
>
> Creates a new `File` instance by converting the given pathname string into an abstract pathname. If the given string is the empty string, then the result is the empty abstract pathname.
>
> **Parameters:**
>
> `pathname` – A pathname string
>
> **Throws:**
>
> `NullPointerException` – If the pathname argument is null

We can tell at once that the `FileWriter` constructor throws a checked exception, because the header contains what is known as a `throws` clause:

```
public FileWriter(File file) throws IOException
```

This shows that the exception thrown is an `IOException`. Further on there is then a paragraph describing when the exception will occur:

> **Throws:**
>
> `IOException` – if the file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason

The `File` constructor also throws an exception, but it is an unchecked exception; we can see this because there is no `throws` clause in the constructor's header. From the paragraph describing the `throws` clause we can see that it is an instance of `NullPointerException` which is thrown. `NullPointerException` is a direct subclass of `RuntimeException`, and you know from *Unit 8* (Subsection 3.2) that a `RuntimeException` is an unchecked exception, which the compiler does not force the programmer to catch.

Reading the documentation tells us that when we use the `FileWriter` constructor we must enclose it in a `try–catch` statement, but the `File` constructor can be used without `try–catch` (although we ought to check that the pathname is not `null`, unless we want to risk a `NullPointerException`!).

Throughout the rest of this unit you need to be aware that constructors and methods of classes in the `java.io` library often throw checked exceptions, which you will need to catch in `try–catch` statements.

2.2 Writing to a file

In Subsection 1.1 you learnt that the `Reader` and `Writer` classes of the `java.io` library handle textual information made up of characters and strings, whereas the `InputStream` and `OutputStream` classes handle binary data.

In this subsection we look at writing characters to a text file, so we will use a subclass of the `Writer` class. Note that the files we produce here are just ordinary files. Not only can we inspect them from Notepad or WordPad, but they can also be edited.

The `FileWriter` class

The simplest `Writer` class you can use to open an output stream to write text to a file is `FileWriter`. However, first we need to create a `File` object to describe a physical file, just as we did in Subsection 1.2:

```
String pathname = OUFileChooser.getFilename();
File aFile = new File(pathname);
```

Now we can create an instance of `FileWriter` using our instance of `File` as an argument to the `FileWriter` constructor with the following signature:

```
FileWriter(File)
```

The constructor will throw an `IOException` if the physical file described by its argument cannot be opened. If the file specified does not exist, but the folder in which it should be found is there, then the physical file will be created and no error is thrown. We are opening the file to write something to it, so it is perfectly reasonable to specify a non-existent file. However, if the file exists, but is marked as read-only then you will again get

an `IOException`. This is why it is sensible on creating the `File` object to test whether it exists (and if so ask the user whether it should be overwritten) and test whether it is read-only (and if so ask the user to choose another file). Although it is prudent to carry out the tests, our examples will not include them since we wish to keep our code as concise as possible, in order to concentrate on the teaching of reading and writing to files.

Because `FileWriter` constructors can throw checked exceptions, the code for the creation of a `FileWriter` object must be written within the `try` block of a `try–catch` statement:

```
try
{
    FileWriter aFileWriter = new FileWriter(aFile);
    // Code to write to the file goes here
}
catch (Exception anException)
{
    // Code to catch any exceptions thrown by the FileWriter constructor
}
```

This would give us the following situation:

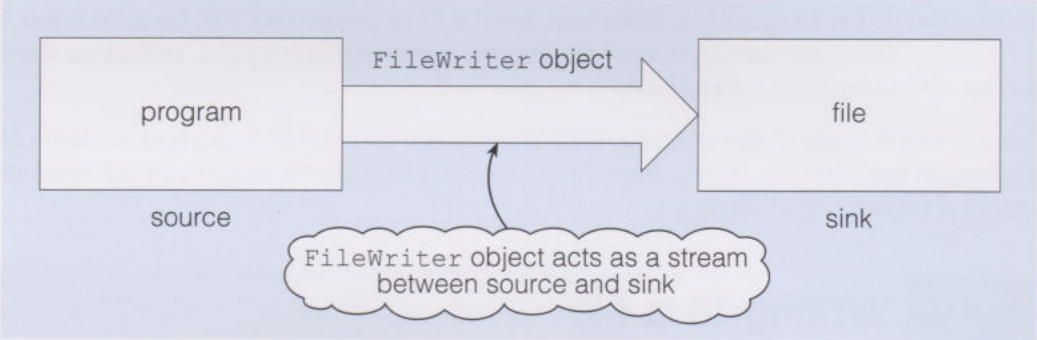


Figure 5 A `FileWriter` object acting as a stream between a source and a sink

Characters or strings can now be written one at a time to the file, via the `FileWriter` stream using `write()` messages, as follows:

```
aFileWriter.write('H');
aFileWriter.write('e');
aFileWriter.write('l');
aFileWriter.write('l');
aFileWriter.write('o');
aFileWriter.write(System.getProperty("line.separator"));
aFileWriter.write("World");
```

The code `System.getProperty("line.separator")` is simply a way of getting a platform-independent line break into the file.

You may be familiar with using the 'newline' escape sequence `\n` to end a line, and it is after all much quicker to add `\n` on to the end of a string than to use an additional statement. However, although `\n` works in the Display Pane, not all applications will interpret it in the way we might expect. For example, suppose you were to use the following statements to write to a file:

```
aFileWriter.write("Writing line 1 to a file.\n");
aFileWriter.write("Writing line 2 to a file.\n");
aFileWriter.write("Writing line 3 to a file.\n");
```

Then if you opened the resultant file in Notepad you would see this:

```
Writing line 1 to a file.□Writing line 2 to a file.□Writing line 3 to a file.□
```

In fact whether `\n` will produce a new line also varies between different operating systems, so if you want your program to be portable it is best to use `System.getProperty("line.separator")`, because it is guaranteed to work correctly. Note that it is a string, so you can assign it to a variable and then concatenate it like any other string, for example:

```
String newLine = System.getProperty("line.separator");
aFileWriter.write("Greetings earthlings." + newLine);
aFileWriter.write("Take me to your leader!");
```

Once the data has been written to the file, the `FileWriter` stream needs to be closed (the reasons for this are explained later). This is easily done with a `close()` message:

```
aFileWriter.close();
```

Putting all the code together from the steps above we get the following:

```
String pathname = OUFileChooser.getFilename();
File aFile = new File(pathname);
try
{
    FileWriter aFileWriter = new FileWriter(aFile);
    aFileWriter.write('H');
    aFileWriter.write('e');
    aFileWriter.write('l');
    aFileWriter.write('l');
    aFileWriter.write('o');
    aFileWriter.write(System.getProperty("line.separator"));
    aFileWriter.write("World");
    aFileWriter.close();
}
catch (Exception anException)
{
    System.out.println("Error: " + anException);
}
```

Finally we will now explain the `catch` block of the `try–catch` statement. The constructor `FileWriter()` can throw instances of `IOException`, yet we have decided to declare the argument to the `catch` block as being of type `Exception`. This means that the `catch` block will catch any exception that is an instance of any subclass of `Exception`, including of course `IOException`. We have done this to keep things simple.

Notice the print statement inside the `catch` block. It is recommended that you use such a print statement in all `catch` blocks, as it will give you both the class of the exception and the message associated with it. For example, if you selected the file `backup.txt` from the file chooser dialogue box, which would cause the `FileWriter()` constructor to throw an exception, the print statement would display the message

```
Error: java.io.FileNotFoundException:
C:\Documents and Settings\<username>\My Documents\M255\M255Projects\
Unit12_Project_1\backup.txt (Access is denied)
```

which in most cases should be sufficient to help you diagnose the problem.

Sometimes programmers are tempted to save effort by using empty `catch` blocks:

```
catch (Exception anException)
{
}
```

This is not wise! If an exception occurs something must have gone wrong, but because the `catch` block contains no print statement there will be no feedback to indicate where or what the problem is.

ACTIVITY 4

Launch BlueJ and open the project Unit12_Project_1. Open the OUWorkspace and following the pattern given above, write statements to do the following.

Use Windows Explorer to check that there is not already a file called firstWriteTest.txt in the Unit12_Project_1 folder. If there is such a file you must either rename it or use a different pathname throughout the activity.

1 Set the pathname of a new file (enter the filename `firstWriteTest.txt` when requested).

2 Create a new `File` object.

3 Within a `try` block, create an instance of `FileWriter`, then:

   (a) using a `write()` message, write the string `"To be or not to be"` to the `FileWriter` stream;

   (b) write a linebreak to the `FileWriter` stream;

   (c) write the string `"That is the question"` to the `FileWriter` stream;

   (d) close the `FileWriter`.

4 Write a `catch` block to catch any exceptions.

Once you have written all the code, select it all and execute it.

You can test that your code worked by examining the contents of the file in one of two ways:

- Select Open from the OUWorkspace's File menu. You will be asked via a dialogue box whether you want to insert the contents of any file you select into the Code Pane or whether you want to replace the contents of the Code Pane with the contents of the file. Choose to insert, and then select your new file – its contents will appear at the bottom of the Code Pane.
- Use Windows Explorer to navigate to the folder containing the Unit12_Project_1 project, and double-click on the file to open it (in most cases the file will be opened in Notepad). Check that it contains the correct text. Close Notepad (or your default text editor if different).

In the final part of this activity you are going to reuse the code you have already written to write once more to the file `firstWriteTest.txt`. Modify your code so that it now writes just the string `"Whether 'tis nobler in the mind"` to the file. Select and execute your code and then check the contents of the file – are the contents what you expected?

DISCUSSION OF ACTIVITY 4

Your code should look similar to this:

```
String pathname = OUFileChooser.getFilename();
File aFile = new File(pathname);
try
{
    FileWriter aFileWriter = new FileWriter(aFile);
    aFileWriter.write("To be or not to be");
    aFileWriter.write(System.getProperty("line.separator"));
    aFileWriter.write("That is the question");
    aFileWriter.close();
}
catch (Exception anException)
{
    System.out.println("Error: " + anException);
}
```

Alternatively, the first line of code could be written like this:

```
String pathname = OUFileChooser.getFilename("firstWriteTest.txt");
```

giving the name of the file directly, and letting the `getFilename()` method supply the rest of the pathname, so avoiding use of the file chooser dialogue box.

You should find that the file contains the following:

```
To be or not to be
That is the question
```

After you have modified your code to write the single string `"Whether 'tis nobler in the mind"` to the file `firstWriteTest.txt`, you will find that the file now contains only that string – the previous contents of the file have been overwritten!

Congratulations! You have just created and written to your first file. You may have been surprised to discover that reopening the file and writing to it again overwrote the previous contents. However, in many situations this is exactly what we want. If we want to append to a file rather than overwrite its previous contents we need to use a different `FileWriter` constructor. Here is its header:

```
public FileWriter(File file, boolean append) throws IOException
```

If the second argument to this constructor is `true`, then characters will be written to the end of the file rather than overwriting the previous contents.

2.3 Reading from a file

In the previous subsection we learnt how to write to a file. In this subsection we look at reading characters from a text file, and to do so we will use a subclass of the `Reader` class.

The `FileReader` class

The simplest `Reader` class you can use to open an input stream to read characters from a file is the `FileReader` class. Just as for writing to a physical file, we first need to create a `File` object to describe the physical file we want to read:

```
String pathname = OUFileChooser.getFilename();
File aFile = new File(pathname);
```

Now we can create an instance of `FileReader` and use our instance of `File` as an argument to the `FileReader` constructor with the following header:

```
FileReader(File file)
```

The constructor will throw a `FileNotFoundException` if the physical file described by its argument `file` does not exist, if it is a directory, or if for some other reason it cannot be opened for reading. Hence code that uses the constructor must be put into the `try` block of a `try–catch` statement:

```
try
{
    FileReader aFileReader = new FileReader(aFile);
    // Code to read from the file goes here
}
catch
{
    // Code to catch any exceptions thrown
}
```

This would give us the following situation:

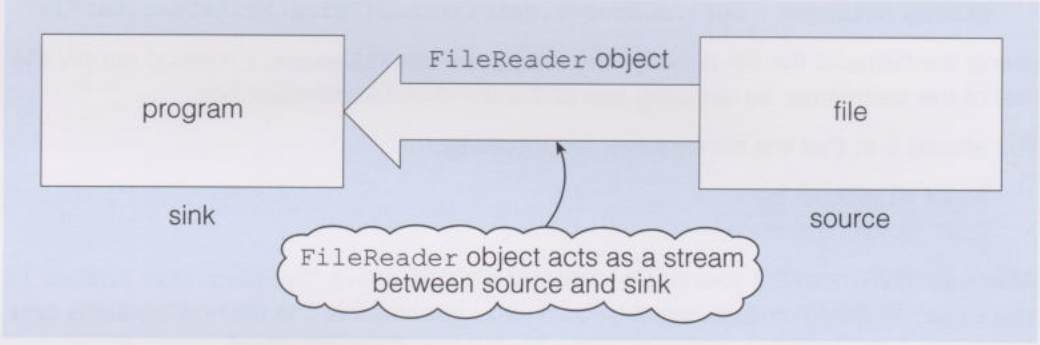


Figure 6 A `FileReader` object acting as a stream between a source and a sink

Characters can now be read one at a time from the file, via the `FileReader` stream using `read()` messages. The `read()` method returns an integer (the ASCII value of the character read from the stream), or `-1` if the end of the stream has been reached. If we wish to print the value returned by `read()` as a character (for example, to the Display Pane) we need to *cast* the integer into a `char`. For example:

```
int ch = aFileReader.read();
System.out.print((char) ch);
```

To read and print out the entire contents of a file we need to use a `while` loop, which will continue looping until `read()` returns `-1` (i.e. the end of the stream has been reached). For example:

```
int ch = aFileReader.read();
while (ch != -1)
{
    System.out.print((char) ch);
    ch = aFileReader.read();
}
```

Finally of course we need to close the stream with

```
aFileReader.close();
```

and of course write the code for the `catch` block, which would be exactly the same as the code for the `FileWriter` examples.

ACTIVITY 5

Launch BlueJ and open the project Unit12_Project_1. Check that the file `firstWriteTest.txt` exists in the folder Unit12_Project_1, and that it contains some characters. Then open the OUWorkspace, and following the pattern given above, write the code to read the contents of the file `firstWriteTest.txt`, printing the contents to the Display Pane.

DISCUSSION OF ACTIVITY 5

Your code should look similar to this:

```
String pathname = OUFileChooser.getFilename();
File aFile = new File(pathname);
int ch;
try
{
    FileReader aFileReader = new FileReader(aFile);
    ch = aFileReader.read();
    while (ch != -1)
    {
        System.out.print((char) ch);
        ch = aFileReader.read();
    }
    aFileReader.close();
}
catch (Exception anException)
{
    System.out.println("Error: " + anException);
}
```

3 Buffering and wrapping classes

In Section 2 we started writing and reading characters to and from files. The classes we used to accomplish this were `FileWriter` and `FileReader`. While these classes seemed to work well for us, they are in fact a very inefficient way to read and write any sizeable amount of data to and from files. This is because reading and writing to physical files on a hard disk (or any other peripheral device) are processor-intensive operations: each write to a `FileWriter` stream involves another subsequent write to the physical file, and every read from a `FileReader` stream involves another read from the physical file. Furthermore a program can write or read data to or from memory (i.e. to or from a stream) far faster than the operating system can (behind the scenes) write or read that data to or from a physical file on a peripheral device. Hence, using `FileReader` and `FileWriter` classes in this way can slow a program down to the speed at which data can be read from and written to the physical device.

So we have a mismatch of capabilities: the speed at which a program can write to or read from a stream (which is relatively fast); and the speed at which the operating system can read from or write to a file on a peripheral device (which is relatively slow). The answer to this is to use what are termed buffers, which are explained in the next subsection.

3.1 Buffers

Among the classes in `java.io` are the `Buffered` classes: `BufferedReader`, `BufferedWriter`, `BufferedInputStream` and `BufferedOutputStream`. Why are these classes important?

You were introduced to the idea of **buffers** in *Unit 9* (Section 5). An instance of `StringBuilder` has an underlying character array which is usually larger than the number of characters held. The 'empty' array positions form a buffer, which is used when the characters held in the `StringBuilder` object are changed.

You can think of a buffer as a mechanism to even out supply and demand. For example, Figure 7 shows a water butt. The butt is filled via a drainpipe. When it rains, water is collected from the gutters of the house and flows down the drainpipe into the water butt. When it is raining, the gardener does not need to water the garden, so the rainwater is stored in the water butt. After a few dry days, the gardener will need to water the garden and the watering can is filled via the tap in the water butt. Unless there is a prolonged dry spell, there will be sufficient water in the water butt for the gardener to water the garden without the water butt running dry. So the water butt is acting as a buffer allowing the storage of water at a time when it is raining and allowing use of the water when it is not.

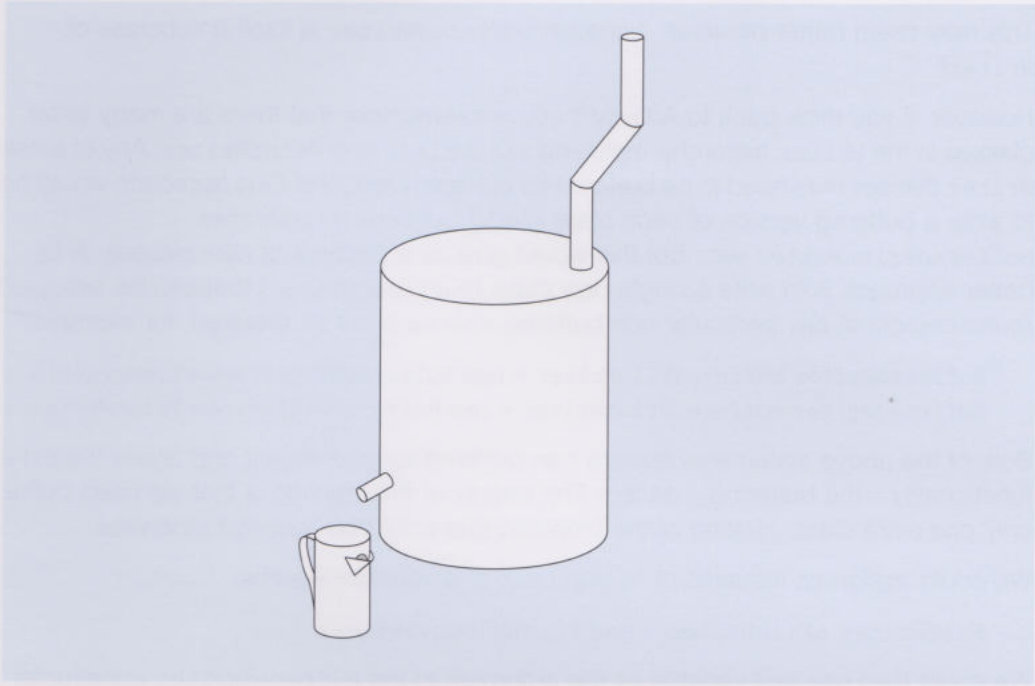


Figure 7 A water butt as a buffer enabling the gardener to even out the supply of and demand for water in a garden

In a similar way, a program may produce data faster than it can be written to a file on a hard disk. Rather than forcing the program to run more slowly, so that data is produced at the same rate at which it can be written, the program can write the data to a buffer, which holds it temporarily until it can be written to the file. In this case, we can see the program as the producer (it is generating the data) and the file as the consumer (it is accepting and storing the data). By using a buffer we are making the producer and consumer more independent of one another, so they can work at different rates or on different-sized chunks of data.

Most programming languages and operating systems use buffers to store data as it is transferred between a program and an external device (e.g. hard disk, CD, DVD, printer or scanner). By using a buffer, the external device can access the buffer in its preferred way and the program can access the buffer as required. This decoupling of the program and external device improves efficiency. In Java this efficiency is gained by using the `Buffered` stream classes (`BufferedReader`, `BufferedWriter`, `BufferedInputStream` and `BufferedOutputStream`) for input and output. For writing and reading text files we shall use the `BufferedWriter` and `BufferedReader` classes. You will see how we do this in the next subsection.

3.2 Wrapping stream classes

You might think that if you wanted to open a `BufferedWriter` to write to physical file, you could do something like the following:

```
String pathname = OUFileChooser.getFilename();
File aFile = new File(pathname);
BufferedWriter aBufferedWriter = new BufferedWriter(aFile);
```

However, you would quickly find that this would not compile. If you examine the documentation for the class `BufferedWriter` you will see that neither of its constructors accepts a `File` argument and both require an argument of type `Writer`, for example:

```
public BufferedWriter(Writer out)
```

This may seem rather perverse, because `BufferedWriter` is itself a subclass of `Writer`!

However, if you think back to Activity 1 you will remember that there are many writer classes in the `Writer` hierarchy, including `FileWriter` and `PrintWriter`. Any of these `Writer` classes may need to be buffered for efficiency reasons. One approach would be to write a buffered version of each class giving `BufferedFileWriter`, `BufferedPrintWriter` etc., but this would give us a plethora of new classes. A far better approach is to write a single new class (`BufferedWriter`) that can be *wrapped round* objects of any particular non-buffered `Writer` class as required, for example:

```
BufferedWriter bufferedFileWriter = new BufferedWriter(new FileWriter(aFile));
BufferedWriter bufferedPrintWriter = new BufferedWriter(new PrintWriter(aFile));
```

Both of the above statements takes a non-buffered `Writer` object and layers the extra functionality – the buffering – on top. The beauty of this solution is that we need define only *one* extra class, instead of the umpteen that would be required otherwise.

We could assign an instance of `FileWriter` to a variable like this:

```
FileWriter aFileWriter = new FileWriter(aFile);
```

We could then use that variable as the argument to the `BufferedWriter` constructor with the signature `BufferedWriter(Writer out)`:

```
BufferedWriter bufferedFileWriter = new BufferedWriter(aFileWriter);
```

But as we never need to refer directly to the `FileWriter` object, it can be created anonymously as an argument to the `BufferedWriter` constructor, as shown in the first example above.

Here is what an instance of `BufferedWriter` might look diagrammatically:

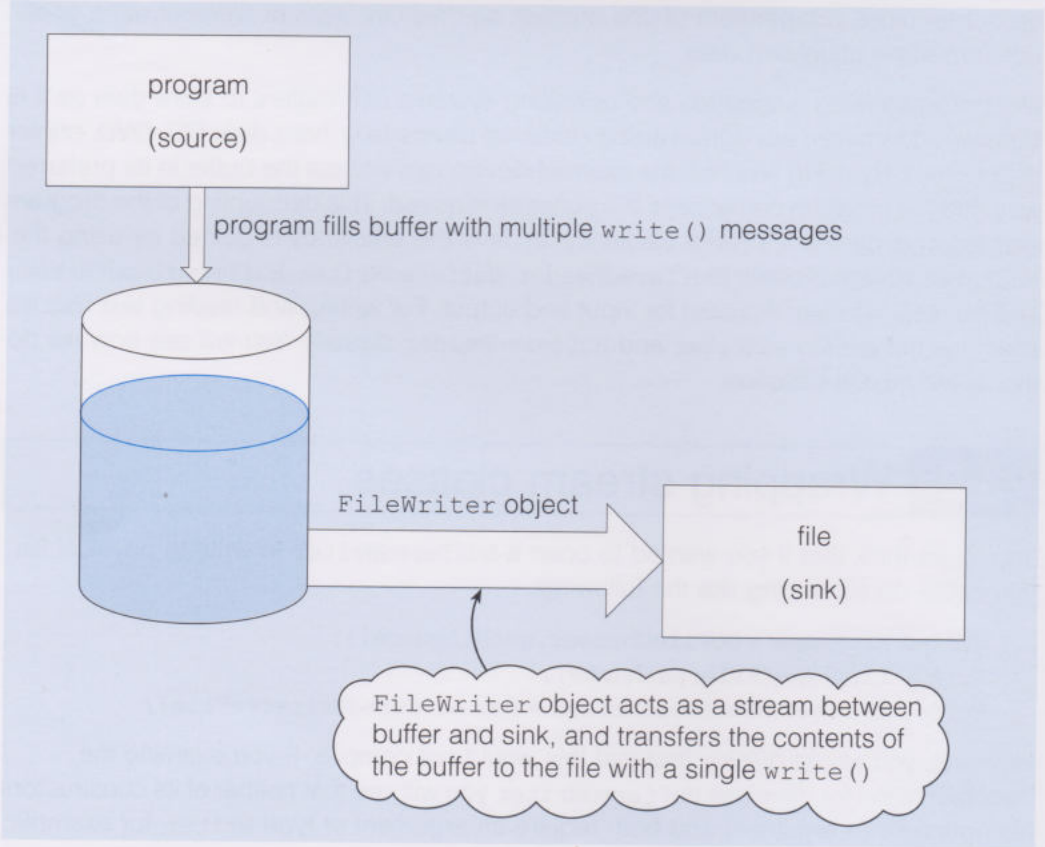


Figure 8 Using a `BufferedWriter` to provide a temporary storage area for data produced by a program before it is written to a file

Now when we want to write to a text file, we write characters or strings to the buffer, then when the buffer is full, or if we explicitly flush it with a `flush()` message, the entire contents of the buffer is written to the `FileWriter` stream as an array of characters. So instead of lots of processor intensive writes to the physical file, we just have one big one, which is much more efficient. Note also that when you send a `close()` message to a `BufferedWriter`, its contents are automatically flushed to the stream before it closes.

Similarly, to read files we can wrap a `FileReader` with a `BufferedReader` as follows:

```
BufferedReader bufferedFileReader = new BufferedReader(new FileReader(aFile));
```

Here is what an instance of `BufferedReader` might look diagrammatically:

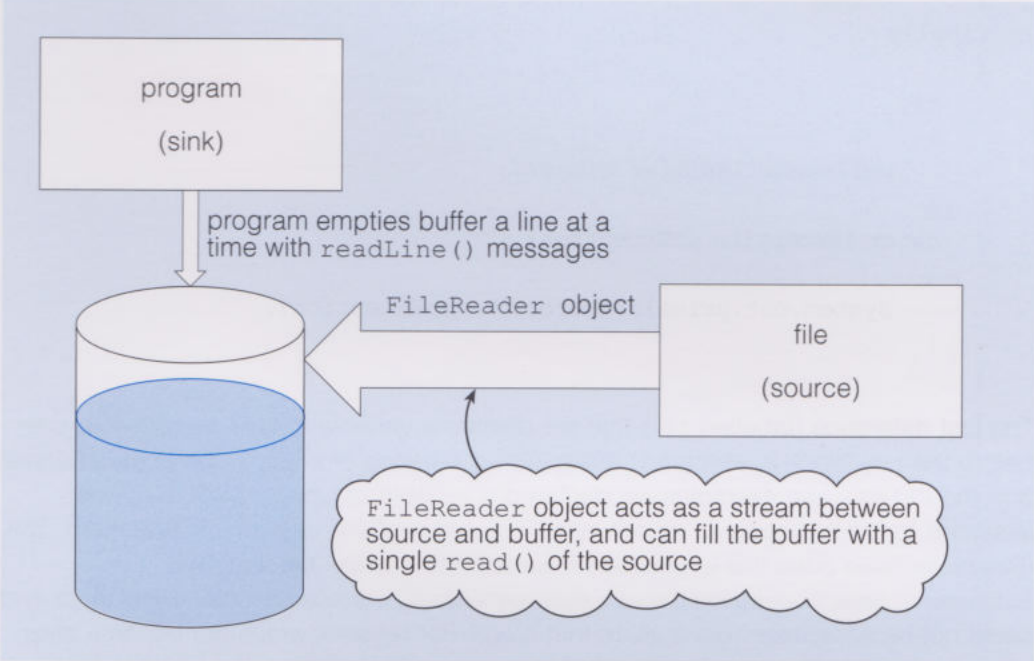


Figure 9 Using a `BufferedReader` to provide a temporary storage area for data from a file before it is read by a program

As well as providing buffering, the classes `BufferedWriter` and `BufferedReader` also define two important and useful methods. `BufferedReader` provides the method `readLine()`, which reads in a whole line of text from the buffer as a string. (Remember when just using a `FileReader`, we could read only one character at a time, as an ASCII value, which we then had to cast into a `char`.) `BufferedWriter` provides the method `newLine()`, which simplifies adding a platform-independent line break to a text file.

The first `readLine()` message sent to an instance of `BufferedReader`, before returning the first line of the file, causes the `FileReader` stream to completely fill the buffer with data from the file, in a single read. Subsequent `readLine()` messages gradually empty the buffer, and it's not until the buffer is empty that the `FileReader` needs to refill the buffer with data from the file.

3.3 Writing to a file using a `BufferedWriter`

In this subsection we shall briefly look at the code needed to use a `BufferedWriter` to efficiently write to a file. At first glance this code looks very similar to the type of code we wrote using the `FileWriter` class, but there are some subtle differences which we have labelled with numbers:

```
String pathName = OUFileChooser.getFilename();
File aFile = new File(pathName);
BufferedWriter bufferedFileWriter = null;                          //1
try
{
   bufferedFileWriter = new BufferedWriter(new FileWriter(aFile)); //2
   bufferedFileWriter.write("Writing to a file");
}
catch (Exception anException)
{
   System.out.println("Error: " + anException);
}
finally                                                            //3
{
   try
   {
      bufferedFileWriter.close();                                  //4
   }
   catch (Exception anException)
   {
      System.out.println("Error: " + anException);
   }
}
```

The first difference (labelled `1`) is that we declare a variable of type `BufferedWriter` before the `try` block in which it is assigned an instance of `BufferedWriter`, whereas with the `FileWriter` examples we declared a variable of type `FileWriter` and assigned to it a `FileWriter` object in the `try` block of the `try–catch` statement. The reason we have done this is because if we were to declare the variable `bufferedFileWriter` inside the `try` statement block, it would be local to that block and could not be accessed from outside that block. For reasons which will become clear when we discuss the line of code labelled `3`, we need to access the variable from outside the `try` block. The other thing to note about this variable declaration is that we have initialised it to `null`. We have done this because otherwise when the compiler came to parse the line labelled `4` it would display the warning message

```
variable bufferedFileWriter might not have been initialized
```

and highlight the line of code labelled `4`. The compiler does this because if at run-time the line labelled `2` caused an exception, `bufferedFileWriter` would indeed not be initialised.

The line labelled `2` is simply the creation of the `BufferedWriter` object, which we discussed at length in Subsection 3.2.

The line labelled `3` introduces a new keyword – `finally`. This is an optional part of a `try–catch` statement which is very important to use when working with streams and files. It provides a cleanup mechanism which executes regardless of what happens within the `try` block. If code within the `try` block throws an exception, the code in the `finally` block will be executed before exception handling passes control to a different part of the program. Hence, `finally` blocks are typically used to guarantee the closure of files or to release other system resources.

You will notice that we close the `BufferedWriter` in the above code at the line labelled `4` within the `finally` block. Although `finally` is semantically part of the `try–catch` block, it has its own scope and is unable to refer to variables that are local to the `try` block, which is why we had to declare our variable `bufferedFileWriter` outside, and before, the `try` block in the line labelled `1`.

Because the `close()` method can itself throw an `IOException` (if for some reason the stream cannot be closed) we need to enclose it in a further `try-catch` statement nested within the `finally` block.

From now on in this unit we will close all streams from within a `finally` block rather than from within the `try` block, as we have been doing previously. We should really have done this with the `FileWriter` and `FileReader` streams too, but we did not want to give you too many new concepts all at once. Consider the following code:

```
try
{
    bufferedFileWriter = new BufferedWriter(new FileWriter(aFile));
    bufferedFileWriter.write("Writing to a file");
    bufferedFileWriter.close();
}
catch (Exception anException)
{
    System.out.println("Error: " + anException);
}
```

If the statement `bufferedFileWriter.write("Writing to a file");` threw an exception, the code `bufferedFileWriter.close();` would not execute, as the `catch` block would execute instead, then all execution would terminate and the file would remain open. You will encounter the ramifications of leaving a file open in Activity 7.

ACTIVITY 6

Launch BlueJ and open the project Unit12_Project_1. Open the OUWorkspace.

Using an instance of `BufferedWriter`, write the following lines of text to a new file called `poem.txt`:

> Hope is the thing with feathers
>
> That perches in the soul
>
> And sings the tune without the words
>
> And never stops at all

Once you have written (and closed!) the file `poem.txt`, check its contents to ensure your code has worked properly.

After you have completed the activity, keep the OUWorkspace open, and immediately do Activity 7, which uses the code that you write for this activity.

DISCUSSION OF ACTIVITY 6

You code should look similar to this:

```
String pathname = OUFileChooser.getFilename("poem.txt");
File aFile = new File(pathname);
BufferedWriter bufferedFileWriter = null;
try
{
    bufferedFileWriter = new BufferedWriter(new FileWriter(aFile));
    bufferedFileWriter.write("Hope is the thing with feathers");
    bufferedFileWriter.newLine();
    bufferedFileWriter.write("That perches in the soul");
    bufferedFileWriter.newLine();
    bufferedFileWriter.write("And sings the tune without the words ");
    bufferedFileWriter.newLine();
    bufferedFileWriter.write("And never stops at all");
}
catch (Exception anException)
{
    System.out.println("Error: " + anException);
}
finally
{
    try
    {
        bufferedFileWriter.close();
    }
    catch (Exception anException)
    {
        System.out.println("Error: " + anException);
    }
}
```

In the next activity you will learn why it is so important to close files.

ACTIVITY 7

Returning to the OUWorkspace and the code you wrote for Activity 6, change the name of the file to be opened to `poem2.txt`:

```
String pathname = OUFileChooser.getFilename("poem2.txt");
```

Then comment out the `try–catch` statement nested in the `finally` block (using `/*` and `*/`) as shown below:

```
finally
{
/*
    try
    {
        bufferedFileWriter.close();
    }
    catch (Exception anException)
    {
        System.out.println("Error: " + anException);
    }
*/
}
```

Next select and execute *all* the code from

```
String pathname = OUFileChooser.getFilename("poem2.txt");
```

to the closing brace of the `finally` block.

When the code has successfully executed, check the contents of `poem2.txt` – what do you find? Is it what you expected?

Close Notepad if you used it to open the file.

Now try to delete `poem2.txt`, clicking on the Yes button when the Confirm File Delete dialogue appears – what happens? Can you explain what has happened?

Now select and execute just the `try–catch` block nested in the `finally` block.

Again check the contents of `poem2.txt`, and then try to delete the file. What do you conclude?

DISCUSSION OF ACTIVITY 7

When you comment out the nested `try–catch` statement within the `finally` block, you find that after the code has been executed the file `poem2.txt` is empty. Then when you try to delete the file you get the following error dialogue box:

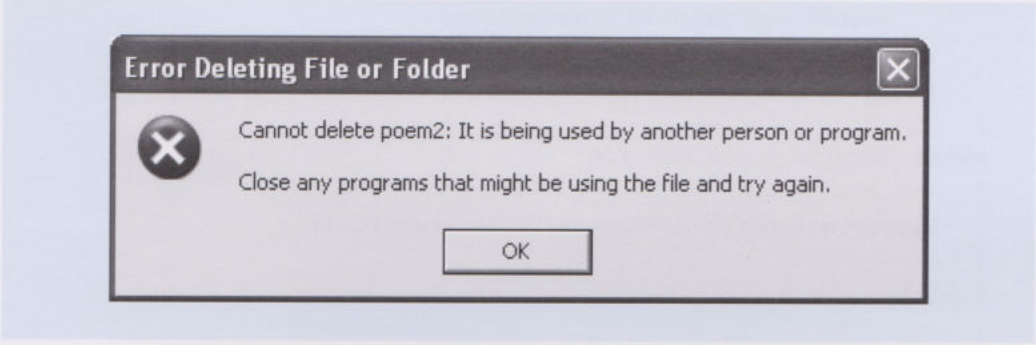


Figure 10 Error dialogue box showing `poem2.txt` is still open

This indicates that the file `poem2.txt` is still open. Once you select and execute the `try–catch` statement nested in the `finally` block and then check the contents of `poem2.txt` once more, you should find that the file is no longer empty and now contains the lines of the poem.

You should also find that the file can be deleted as you would expect.

It may not have surprised you to discover that the file could not be deleted until it had been closed, but why was there no data in it when the write statements had been executed? The answer is that the output is buffered and the characters you have written via the `BufferedWriter` are still in the buffer. Closing the file *flushes* any characters remaining in the buffer to the file before it is closed, ensuring that all the information is written.

3.4 Reading from a file using a `BufferedReader`

The basic pattern for reading from a text file using a `BufferedReader` is very similar to the code we wrote for reading from a file using a `FileReader`, the differences being:

1 you must wrap an instance of `FileReader` with an instance of `BufferedReader`;
2 you read lines of text from the buffer with `readLine()` messages which you put within a `while` loop which terminates when `readLine()` returns `null`;
3 just as we did with an instance of `BufferedWriter`, you should close a `BufferedReader` from a `try–catch` statement nested within a `finally` block.

ACTIVITY 8

If they are not already open, launch BlueJ, open the project Unit12_Project_1, and then open the OUWorkspace. Using an instance of `BufferedReader`, open the file `poem.txt` that you created in Activity 6, and print the contents to the Display Pane.

DISCUSSION OF ACTIVITY 8

Your code should look similar to this:

```
String pathname = OUFileChooser.getFilename("poem.txt");
File aFile = new File(pathname);
BufferedReader bufferedFileReader = null;
try
{
    String currentLine;
    bufferedFileReader = new BufferedReader(new FileReader(aFile));
    currentLine = bufferedFileReader.readLine();
    while (currentLine != null)
    {
        System.out.println(currentLine);
        currentLine = bufferedFileReader.readLine();
    }
}
catch (Exception anException)
{
    System.out.println("Error: " + anException);
}
finally
{
    try
    {
        bufferedFileReader.close();
    }
    catch (Exception anException)
    {
        System.out.println("Error: " + anException);
    }
}
```

4 Making objects persistent using text files

So far in this unit we have written text to files and read text from files. In this section we will continue to work with text files, but in a way that will allow us to make objects (specifically `Account` objects) persistent. By making an object **persistent**, we mean saving the state of an object to a file on non-volatile storage, such as a hard disk, in such a way that the file can be read back into memory to recreate that object.

4.1 Writing the details of `Account` objects to file

To facilitate writing and reading the state of `Account` objects to and from files we will develop a utility class called `AccountsIO`. But first, in the next activity, we will refresh your knowledge of `Account` objects and how to iterate over collections, something you will need to do when developing the `AccountsIO` class.

ACTIVITY 9

Launch BlueJ and open the project Unit12_Project_2. You can see that this contains the familiar `Account` and `CurrentAccount` classes. Open the OUWorkspace and from its File menu open the text file `Activity9.txt`. This will load into the Code Pane the statements needed to create a number of `Account` objects and add them to a set. Select and execute these statements to create the set of `Account` objects.

Your task in this activity is to write the code to iterate over the set of `Account` objects (referenced by `accountSet`), printing details about the state of the objects (using `System.out.println()`) to the Display Pane. Your code should first print out a string describing what the output means, followed by the details for each account, like this:

Sets were introduced in *Unit 10*.

```
Account Details (Holder, Account Number and Balance)
John Smith 020 150.0
David Green 010 50.0
Mary Jones 030 175.5
```

Note that because the `Account` objects are held in a set, and sets have no particular order associated with them, your output to the Display Pane may print out the state of the accounts in a different order.

DISCUSSION OF ACTIVITY 9

Your code should look similar to the following:

```
System.out.println("Account Details (Holder, Account Number and
Balance)");
for (Account eachAccount : accountSet)
{
    System.out.println(eachAccount.getHolder() + " "
                       + eachAccount.getNumber() + " "
                       + eachAccount.getBalance());
}
```

Now you have refreshed your memory of sets, accounts and iteration, it is time to start developing the `AccountsIO` class that we will use to write the details of account objects to file and then read those details back to recreate the original account objects.

ACTIVITY 10

If it is not already open, Launch BlueJ and open the project Unit12_Project_2.

Click on the New Class button on the left-hand side of the BlueJ window and, when prompted, type in the class name `AccountsIO`. When the icon for the new class appears in the BlueJ window, double-click on it to open the editor.

You will need to import the collection classes from the library `java.util`, the stream classes from the library `java.io` and the class `OUFileChooser` from the library `ou`. So at the top of the class file insert the following `import` statements:

```
import java.util.*;
import java.io.*;
import ou.*;
```

The next step is to write a *class* method called `generateReport()` to write the details of a collection of `Account` objects to a file using an instance of `BufferedWriter`. Here is the method heading:

```
/**
 * Prompts the user for a pathname and then attempts to open a stream
 * on the file specified by the pathname. The method then writes
 * the details of the accounts held in the argument accountCollection
 * to the stream. The account details are preceded by an explanatory
 * heading explaining the order of the information.
 */
public static void generateReport(Collection<Account> accountCollection)
```

Here is how you should tackle writing the method.

1 Prompt the user for a file name using a file chooser dialogue box, and use the returned pathname to create an instance of the `File` class.

2 Declare a variable of type `BufferedWriter` named `bufferedFileWriter`, and initialise it to `null`.

3 Within a `try` block, create an instance of `BufferedWriter` that wraps an instance of `FileWriter` and assigns it to the variable `bufferedFileWriter`. Next, using a `write()` message to `bufferedFileWriter`, write out to the stream what the output means, that is:

```
Account Details (Holder, Account Number and Balance)
```

Follow this with a `newLine()` message.

Then iterate over the set of accounts referenced by the instance variable `accountCollection`, in a manner similar to Activity 9, but instead of sending a `println()` message to `System.out` send a `write()` message to `bufferedFileWriter`, followed by a `newLine()` message.

4 Write a `catch` block to catch any exceptions.

5 Write a `finally` block with a nested `try–catch` statement to close the `BufferedWriter`.

Once you have got your class to compile, open the OUWorkspace. From its File menu open the text file `Activity10.txt`, which will load into the Code Pane the code needed to test your `generateReport()` method. Once the code appears in the Code Pane, select and execute the code. When prompted for a file name by the file chooser dialogue box, enter the name `accountsReport.txt`. Finally check the contents of the text file written by `generateReport()` (using Notepad) to ascertain that everything has worked correctly.

DISCUSSION OF ACTIVITY 10

The code for the class `AccountsIO` should be similar to the following (we have deleted the default constructor that BlueJ automatically inserts):

```
import java.util.*;
import java.io.*;
import ou.*;
/**
 * AccountsIO is a utility class that uses static methods to
 * read from and write to text files the details of Account objects.
 *
 * @author M255 Course Team
 * @version 1.0
 */
public class AccountsIO
{
    public static void generateReport(Collection<Account> accountCollection)
    {
        String pathName = OUFileChooser.getFilename();
        File accountFile = new File(pathName);
        BufferedWriter bufferedFileWriter = null;
        try
        {
            bufferedFileWriter = new BufferedWriter(new FileWriter(accountFile));
            bufferedFileWriter.write
                    ("Account Details (Holder, Account Number and Balance)");
            bufferedFileWriter.newLine();
            for (Account eachAccount : accountCollection)
            {
                bufferedFileWriter.write(eachAccount.getHolder() + " "
                                         + eachAccount.getNumber() + " "
                                         + eachAccount.getBalance());
                bufferedFileWriter.newLine();
            }
        }
        catch (Exception anException)
        {
            System.out.println("Error: " + anException);
        }
        finally
        {
            try
            {
                bufferedFileWriter.close();
            }
            catch (Exception anException)
            {
                System.out.println("Error: " + anException);
            }
        }
    }
}
```

The method `generateReport()` should produce a text file containing the following:

```
Account Details (Holder, Account Number and Balance)
John Smith 020 150.0
David Green 010 50.0
Mary Jones 030 175.5
```

Note that because the collection used as an argument to `generateReport()` is a set, and sets have no particular order associated with them, the text in your file may show the state of the accounts in a different order. This text file is designed to be read by a person. We could work out ways to present the information in neatly aligned columns etc., but for now it is sufficient to allow us to see the details of the accounts held.

If you had any problems with Activity 10, the class `AccountsIO` *as developed so far has been added to Unit12_Project_3.*

4.2 Writing files that can be used to recreate objects

In the previous activity you created a new utility class, `AccountsIO`, and wrote the static method `generateReport()`, which creates a text file which exactly reflects the current state of a collection of `Account` objects given as its argument. So in some sense it makes the data represented by those account objects **persistent**. What we look at in the remainder of this section is what we need to do to create a text file that when subsequently read programmatically can be used to recreate those account objects.

Exercise 2

Look again at the text written to the file generated by `generateReport()`. Can you see any problems with using this data file to recreate, programmatically, the account objects it describes? How do you think they might be overcome?

Solution

The first line in the file is:

```
Account Details (Holder, Account Number and Balance)
```

This is an informal description of the account details given in the subsequent lines. The information helps human users understand the data that follows but is not sufficiently precise to allow a program to read the subsequent lines to recreate the account objects.

Although a program could be written to recognise that each line in the file represents an account object it needs to recreate, there is a problem. Each line of the report contains four or five groups of characters separated by a space, and the program would have no way of knowing if an account holder's full name consisted of two, three or more names; so it would have no way of knowing whether the third group of characters represents part of a holder's name or an account number.

This problem arises because we are using the space character in two different ways: to separate individual names within an account holder's full name; and to delimit (separate) the values of the different instance variables within the account object. The solution to this problem is to use an alternative character to delimit the values of the different instance variables. One of the standard delimiters used in files is the comma, and the delimited values are called 'tokens'.

The discussion of Exercise 2 would suggest that the first line of the file, which describes how the data that represents the account objects is laid out in the file, is of no use to a program which needs to use the data to recreate the objects, and that instead the file should have the following structure:

```
John Smith,020,150.0
David Green,010,50.0
Mary Jones,030,175.5
```

The description of each account object is on a separate line, just as before, but the value of each instance variable is separated from the next by a comma. A file in this format is known as a CSV or **comma-delimited file**. 'CSV' stands for either comma-separated variables or comma-separated values, and we say that each comma delimits neighbouring **tokens**. So in the line

```
John Smith,020,150.0
```

the tokens are `"John Smith"`, `"020"` and `"150.0"`.

ACTIVITY 11

If necessary launch BlueJ and open the project Unit12_Project_3, to which the `AccountsIO` class as so far developed has been added.

In this activity you are going to write a static method called `saveAccounts()`. Here is the method comment and header:

```
/**
 * Prompts the user for a pathname and then attempts to open a stream
 * on the file specified by the pathname. The method then writes
 * the details of the accounts held in the argument accountCollection
 * to the stream. The account details are written in CSV format.
 */
public static void saveAccounts(Collection<Account> accountCollection)
```

To write this method, copy and paste `generateReport()` and change the method name to `saveAccounts()`; then simply change the code

```
bufferedFileWriter.write(eachAccount.getHolder() + " "
                         + eachAccount.getNumber() + " "
                         + eachAccount.getBalance());
```

so that it writes the data to file in CSV format.

Test the `saveAccounts()` method by loading and adapting the code given in `Activity10.txt`. When prompted for a file name by the file chooser dialogue box, enter the name `savedAccounts.txt`.

Finally check the contents of the text file written by `saveAccounts()` using Notepad, to ascertain that everything has worked correctly.

DISCUSSION OF ACTIVITY 11

The method `saveAccounts()` is virtually the same as `generateReport()`. The only changes are to the first `try` block, which should now look like this:

```
try
{
    bufferedFileWriter = new BufferedWriter(new FileWriter(accountFile));
    for (Account eachAccount : accountCollection)
    {
        bufferedFileWriter.write(eachAccount.getHolder() + ","
                                 + eachAccount.getNumber() + ","
                                 + eachAccount.getBalance());
        bufferedFileWriter.newLine();
    }
}
```

When you open `savedAccounts.txt` using Notepad you should see (though perhaps in a different order)

```
John Smith,020,150.0
David Green,010,50.0
Mary Jones,030,175.5
```

with the details of each account on a separate line, and the values of the instance variables delimited by commas.

If you had any problems with Activity 11, the class `AccountsIO` *as developed so far has been added to Unit12_Project_4.*

4.3 Reading a text file using the `Scanner` class

In the previous subsection you created a text file that held details of the states of `Account` objects in CSV format. In this subsection you will learn how to read such a file to recreate the described objects.

From your experiences of reading files in Section 3, you might imagine that we would simply use an instance of the `BufferedReader` class to read a text file saved in CSV format, in order to recreate the objects represented by the data in the file. Well we could, and we might, but then we would have to think of how to programmatically chop up each line that was read, into the comma delimited tokens – this would be tedious. Fortunately the library `java.util` provides a class called `Scanner` that will do all the hard work for us.

ACTIVITY 12

If it is not already open, launch BlueJ. From the Help menu select Java Class Libraries. When your browser opens, select `java.util` from the top-left scrollable frame. Then from the bottom-left scrollable frame select the `Scanner` class.

Explore the documentation for the `Scanner` class and try to find some constructors and methods that might help you to read a text file in CSV format and split each line into its various tokens.

DISCUSSION OF ACTIVITY 12

You should have discovered that the `Scanner` class has many constructors. You may have found the following constructor:

```
public Scanner(Readable source)
```

You might also have discovered that the `BufferedReader` class implements the `Readable` interface, and so this constructor would be useful in conjunction with an instance of `BufferedReader`.

We have been working with strings a lot in this unit, so the following constructor may have caught your eye:

```
public Scanner(String source)
```

The following method may also have caught your attention:

```
public boolean hasNextLine()
```

It returns `true` if there is another line in the source (the argument given to the constructor) and `false` otherwise – such a method would be useful to control a `while` loop.

The method

```
public String nextLine()
```

returns the next line in a source – so this method would be useful for getting the next line in a multi-line source such as a `File`, `FileReader` or `BufferedReader`.

We have been discussing CSV files, and how commas can be used to delimit tokens. So the following method looks useful:

```
public Scanner useDelimiter(String pattern)
```

It informs a `Scanner` object about the character(s) used to delimit tokens in the source.

The method

```
public boolean hasNext()
```

returns `true` if the there is another token in the source, so again this looks useful for controlling a `while` loop.

The method

```
public String next()
```

returns the next token in the source as a string – this looks very promising.

Another method that might prove useful is:

```
public double nextDouble()
```

It returns the next token in the source as a `double`, but only if the characters in the token can be interpreted as a `double` value.

From the previous activity you should have deduced that a `Scanner` object can be used to parse a source such as a `BufferedReader` or a `String` into its constituent tokens. In the next activity you will parse a string to print out the constituent components.

ACTIVITY 13

Note that there are no spaces surrounding the commas as otherwise the spaces would be interpreted as being part of the tokens.

In this activity you will investigate using a `Scanner` object to break down a string into its constituent tokens. Launch BlueJ and open the OUWorkspace.

Enter the following code in the OUWorkspace:

```
Scanner aScanner = new Scanner("David Green,010,50.0");
while (aScanner.hasNext())
{
    System.out.println(aScanner.next());
}
```

Note how the message-send `aScanner.hasNext()` is used to control the `while` loop. The `while` loop will continue to execute until `hasNext()` returns `false`, i.e. when there are no more tokens in the source. Note also how the `next()` message is used within the `while` loop to return the next token in the source as a string.

Select and execute the above code, and observe the output in the Display Pane.

1 How many tokens are printed to the Display Pane?

2 What do you think is the default delimiter of a `Scanner`?

Now alter the code to include the statement `aScanner.useDelimiter(",");` as shown below:

```
Scanner aScanner = new Scanner("David Green,010,50.0");
aScanner.useDelimiter(",");
while (aScanner.hasNext())
{
    System.out.println(aScanner.next());
}
```

Select and execute the modified code, and observe the output in the Display Pane.

3 How many tokens are printed to the Display Pane this time?

DISCUSSION OF ACTIVITY 13

1 Two tokens are printed to the Display Pane. If there are more than two lines of text in your Display Pane, check that you did not include any spaces around the commas in the input text.

2 The string is split on the basis of the space character. The default delimiter of a `Scanner` is one or more whitespace characters. Whitespace characters include space, tab and newline characters.

3 Three tokens are returned: the tokens are now delimited by a comma rather than a space.

In the previous activity you saw how the message `next()` when sent to a `Scanner` object returned the next token in the scanner's source as a string. However, we may not want the next token returned as a string, as the information held in the token may be a character representation of a different type. Fortunately the class `Scanner` provides a number of methods which will return the next token as a value of another type. For example, `nextInt()` will return the characters comprising the next token as an `int` value, so long as those characters can be interpreted as an `int` value – if they cannot, an exception will be thrown. Similarly `nextDouble()` will return the characters comprising the next token as a value of type `double`.

ACTIVITY 14

Unless it is already open, launch BlueJ and open the project Unit12_Project_4. Then open the OUWorkspace.

1 Use a `Scanner` to parse the source string

```
"David Green,010,50.0"
```

so that the extracted tokens are assigned to three variables, declared as follows:

```
String accountHolder;
String accountNumber;
double accountBalance;
```

Obviously the first two tokens you parse must be returned as strings and the third must be returned as a `double` value. (Hint: you do not need a `while` loop to achieve this.)

Once you have parsed the string and assigned values to the three variables, use these variables as the arguments to an `Account` constructor to create and initialise an `Account` object, which should be assigned to a variable named `anAccount`.

Once you have done this, inspect `anAccount` to ascertain that the `Account` object has been correctly and successfully created.

2 Repeat the above activity, but this time using the string

```
"David Green,010,Fifty"
```

DISCUSSION OF ACTIVITY 14

1 Your code should be similar to the following:

```
String accountHolder;
String accountNumber;
double accountBalance;
Account anAccount;
Scanner aScanner = new Scanner("David Green,010,50.0");
aScanner.useDelimiter(",");
//return the next token as a string
accountHolder = aScanner.next();
// return the next token as a string
accountNumber = aScanner.next();
// return the next token as a double
accountBalance = aScanner.nextDouble();
anAccount = new Account(accountHolder, accountNumber, accountBalance);
```

Inspecting `anAccount` should show that:

- `accountHolder` is set to `"David Green"`;
- `accountNumber` is set to `"010"`;
- `accountBalance` is set to `50.0`.

2 When you repeated the activity with the string `"David Green,010,Fifty"`, the following error message should have appeared in the Display Pane:

```
Exception: line 12. java.util.InputMismatchException
```

The statement at line 12 is:

```
accountBalance = aScanner.nextDouble();
```

What has happened is that `nextDouble()` is trying to convert the third token into a `double` value but the third token consists of the characters `F i f t y`, which cannot be converted to a `double`. Hence there is a mismatch between the expectation of `nextDouble()` (it is looking for numeric characters) and what it finds (the alphabetic characters `F i f t y`), and so the exception is thrown and no account is created.

Such exceptions can be avoided, as the `Scanner` class includes methods such as `hasNextInt()` and `hasNextDouble()` which check whether the characters in the next token are of the expected type. However, this of course would lead to lengthier, more complicated code.

The previous activity has demonstrated how a scanner can be used to parse a string into its constituent tokens. In Activity 8 you wrote code in the OUWorkspace to read the contents of a file using a `BufferedReader` and printed the contents line by line to the Display Pane. In Activity 11 you wrote a static method for `AccountsIO` that saved the details of a collection of `Account` objects as characters in a text file in CSV format. In the next activity you are going to bring this all together by completing a static method for the `AccountsIO` class that will read a file containing the details of accounts in CSV format and return a collection of `Account` objects constructed from those details.

ACTIVITY 15

If necessary launch BlueJ and open the project Unit12_Project_5. Double-click on the `AccountsIO` class to open the editor. Scroll to the end of the file to find the static method `loadAccounts()`. This is an incomplete method based largely on Activity 8, and we have indicated with numbered comments where you need to add code.

1 Declare a variable of type `Set` and assign to it an instance of class `HashSet`.

2 Next, declare three variables as follows:

```
String accountHolder;
String accountNumber;
double accountBalance;
```

These will be used for the tokens that represent `Account` attribute values.

3 Declare a variable of type `Scanner` called `lineScanner`.

4 Create an instance of `Scanner` using the string object referenced by the variable `currentLine` as the argument to the constructor. Assign the `Scanner` object to the variable `lineScanner`.

5 Send a message to `lineScanner` to tell it that its source uses commas for the token delimiters.

6 Just as you did in Activity 14, use `lineScanner` to parse the source string so that the extracted tokens are assigned to the three variables `accountHolder`, `accountNumber` and `accountBalance`.

7 Once you have parsed the source string and assigned values to the three variables, use these variables as the arguments to an `Account` constructor to create and initialise an `Account` object, which should be added to the set `accountSet`.

8 Finally, where indicated in the class file, return `accountSet`.

Once you have got `AccountsIO` to recompile, test your `loadAccounts()` method in the OUWorkspace by executing the following code:

```
Set accounts = AccountsIO.loadAccounts();
```

When prompted for a file name, select the file `Activity15_test1.txt`, which holds details of accounts in CSV format. After the code has executed, inspect the variable `accounts` to ascertain that the set it references contains three `Account` objects. If there is a problem you will need to debug your code!

Once you are satisfied that your `loadAccounts()` method is working correctly, try it out with the following files which have been designed to cause problems!

- `Activity15_test2.txt`
- `Activity15_test3.txt`

After each test, open the file with Notepad to look at its the contents, and then inspect `accounts`. Does the set referenced by `accounts` contain `Account` objects that match those described in the CSV file? Explain any error messages shown in the Display Pane.

DISCUSSION OF ACTIVITY 15

Here is the code for the method `loadAccounts()`:

```
public static Collection<Account> loadAccounts()
{
    String pathname = OUFileChooser.getFilename();
    File aFile = new File(pathname);
    BufferedReader bufferedFileReader = null;
    Set<Account> accountSet = new HashSet<Account>();
    try
    {
        String accountHolder;
        String accountNumber;
        double accountBalance;
        Scanner lineScanner;
        bufferedFileReader = new BufferedReader(new FileReader(aFile));
        String currentLine = bufferedFileReader.readLine();
        while (currentLine != null)
        {
            lineScanner = new Scanner(currentLine);
            lineScanner.useDelimiter(",");
            accountHolder = lineScanner.next();
            accountNumber = lineScanner.next();
            accountBalance = lineScanner.nextDouble();
            accountSet.add(new Account(accountHolder, accountNumber, accountBalance));
            currentLine = bufferedFileReader.readLine();
        }
    }
    catch (Exception anException)
    {
        System.out.println("Error: " + anException);
    }
    finally
    {
        try
        {
            bufferedFileReader.close();
        }
        catch (Exception anException)
        {
            System.out.println("Error: " + anException);
        }
    }
    return accountSet;
}
```

When `loadAccounts()` is tested with the file `Activity15_test1.txt`, the accounts are generated correctly from the file descriptions.

When `loadAccounts()` is tested with the file `Activity15_test2.txt`, the set referenced by `accounts` contains only one `Account` object, and the message

```
java.util.InputMismatchException
```

appears in the Display Pane. Reading the file `Activity15_test2.txt` with Notepad, you can see that in the second line of the file, the third token does not contain characters that could represent a value of type double, so the exception `InputMismatchException` is thrown.

When `loadAccounts()` is tested with the file `Activity15_test3.txt`, the set referenced by `accounts` contains only two `Account` objects, and the message

```
java.util.NoSuchElementException
```

appears in the Display Pane. Reading the file `Activity15_test3.txt` with Notepad, you can see that second line contains only two tokens, so the `NoSuchElementException` is thrown.

If you had any problems with Activity 15, the class `AccountsIO` *as developed so far has been added to Unit12_Project_6.*

In the previous activity you used a combination of a `BufferedReader` to read lines of code from a file and a `Scanner` to then parse each line to extract the tokens. In Activity 12, where you looked at the documentation for the `Scanner` class, you saw that it had a constructor that took an argument of the interface type `Reader`. As the `BufferedReader` class implements the `Readable` interface, you could create an instance of `Scanner` whose source was a `BufferedReader`, i.e. you could wrap a `BufferedReader` with a `Scanner`, just as you have previously wrapped a `FileReader` with a `BufferedReader`. For example:

```
bufferedScanner = new Scanner(new BufferedReader(new FileReader(aFile)));
```

The above line of code creates a `Scanner` object which wraps a `BufferedReader` which in turn wraps a `FileReader`. So why might we wish to wrap a `BufferedReader` with a `Scanner`? The answer is for the additional protocol – the `Scanner` class provides some useful methods which could be used to simplify reading from the buffer.

In particular the `Scanner` class has an instance method called `hasNextLine()` which would have simplified the `while` loop in the `loadAccounts()` method, as it returns `true` or `false` depending on whether there are any more lines in the source. For example, in `loadAccounts()` we had to read the first line from the buffer before entering the `while` loop, the `while` loop then tested that `readLine()` had not returned `null`, and then the last line in the `while` loop read a line from the buffer again, as shown (in outline) below:

```
currentLine = bufferedFileReader.readLine();
while (currentLine != null)
{
    ...
    currentLine = bufferedFileReader.readLine();
}
```

If we had wrapped the `BufferedReader` with a `Scanner` we could have constructed the `while` loop as follows:

```
while (bufferedScanner.hasNextLine())
{
    currentLine = bufferedScanner.nextLine();
    ...
}
```

The message-send `bufferedScanner.hasNextLine()` returns `true` if there is another line of text; otherwise it returns `false`. So the `while` loop will continue as long as there are lines of text to be read. Now there is no need to read the first line from the buffer before entering the `while` loop, and reading a line from the buffer occurs in only one place – as the first line of the `while` block.

ACTIVITY 16

If necessary launch BlueJ and open the project Unit12_Project_6. Double-click on the `AccountsIO` class to open the editor. Modify the `loadAccounts()` method so that it now uses two `Scanner` objects: one that wraps a `BufferedReader`, as described above, to read lines from the file, and another, created in the body of the `while` loop, to parse the current line (returned by the first scanner) for the tokens to create an account (just as you did in Activity 15). Do not forget to close the scanner that wraps the `BufferedReader` in the `finally` block, in order to close the underlying stream.

DISCUSSION OF ACTIVITY 16

Here is our code for the modified `loadAccounts()` method:

```
public static Collection<Account> loadAccounts()
{
    String pathname = OUFileChooser.getFilename();
    File aFile = new File(pathname);
    Scanner bufferedScanner = null;
    Set<Account> accountSet = new HashSet<Account>();
    try
    {
        String accountHolder;
        String accountNumber;
        double accountBalance;
        Scanner lineScanner;
        String currentLine;
        bufferedScanner = new Scanner(new BufferedReader(new FileReader(aFile)));
        while (bufferedScanner.hasNextLine())
        {
            currentLine = bufferedScanner.nextLine();
            lineScanner = new Scanner(currentLine);
            lineScanner.useDelimiter(",");
            accountHolder = lineScanner.next();
            accountNumber = lineScanner.next();
            accountBalance = lineScanner.nextDouble();
            accountSet.add(new Account(accountHolder, accountNumber, accountBalance));
        }
    }
```

```
        catch (Exception anException)
        {
            System.out.println("Error: " + anException);
        }
        finally
        {
            try
            {
                bufferedScanner.close();
            }
            catch (Exception anException)
            {
                System.out.println("Error: " + anException);
            }
        }
        return accountSet;
    }
```

If you had any problems with Activity 16, the completed class `AccountsIO` *has been added to Unit12_Project_7.*

4.4 Summary of stream classes, messages and exceptions

Writing to a text file

1 Classes that make use of the stream classes need to import the `java.io` library:

```
import java.io.*;
```

2 In M255 you can get the pathname for a physical file name by invoking one of the `getFilename()` methods on the `OUFileChooser` class, for example:

```
String pathname = OUFileChooser.getFilename();
```

3 When creating a `File` object, the pathname used as the argument to the constructor need not describe an existing physical file if the intention is to create a new physical file.

```
File aFile = new File(pathname);
```

4 Writing to a physical file is achieved via a stream. For text files the stream should be an instance of `FileWriter`, which for efficiency should be wrapped by an instance of `BufferedWriter`. For example:

```
bufferedFileWriter = new BufferedWriter(new FileWriter(aFile));
```

If the `FileWriter` fails to open or write to `aFile` it throws an `IOException`, which must be caught. Therefore the code that creates the stream and writes to the stream must be within the `try` block of a `try–catch–finally` statement.

5 Writing to a `FileWriter` stream buffered by an instance of `BufferedWriter` is achieved by sending `write()` and `newLine()` messages to the `BufferedWriter`. For example:

```
bufferedFileWriter.write("String data to be written");
bufferedFileWriter.newLine(); // Start a new line
```

Putting it all together we have:

```
try
{
    bufferedFileWriter = new BufferedWriter(new FileWriter(aFile));
    bufferedFileWriter.write("String data to be written");
    bufferedFileWriter.newLine(); // Start a new line
}
catch (Exception anException)
{
    System.out.println("Error " + anException);
}
```

6 The stream is closed by sending a `close()` message to the `BufferedWriter`, which in turn will close the underlying stream (an instance of `FileWriter`). The stream must be closed from the `finally` block of the `try–catch–finally` statement. As closing the stream can throw an exception, the `finally` block has a nested `try–catch` statement. For example:

```
finally
{
    try
    {
        bufferedFileWriter.close();
    }
    catch (Exception anException)
    {
        System.out.println("Error " + anException);
    }
}
```

Reading from a text file using a `Scanner`

1 Classes that make use of the `Scanner` class need to import the `java.util` library:

```
import java.io.*;
import java.util.*;
```

2 In M255 you can get the pathname for a physical file name by invoking the method `getFilename()` on the `OUFileChooser` class, for example:

```
String pathname = OUFileChooser.getFilename();
```

3 A `File` object can then be created as follows:

```
File aFile = new File(pathname);
```

4 Reading a physical file is achieved via a stream. For text files the stream should be an instance of `FileReader`, which for efficiency should be wrapped by an instance of `BufferedReader`. In order to easily read more than one character at a time from the buffer, it is a good idea to wrap the `BufferedReader` with an instance of `Scanner`. For example:

```
bufferedScanner = new Scanner (new BufferedReader(new FileReader (aFile)));
```

If the `FileReader` fails to open or read `aFile` it throws an `IOException`, which must be caught. Therefore the code that creates the stream and reads from the stream must be within the `try` block of a `try–catch–finally` statement.

5 The message `hasNextLine()` when sent to a `Scanner` will return `true` if there are any more lines of characters in the underlying stream, and `false` otherwise. The message `nextLine()` when sent to a `Scanner` will return the next line of characters from the stream as a string.

Putting it all together we have:

```
try
{
    bufferedScanner = new Scanner (new BufferedReader(new FileReader (aFile)));
    String line;
    while (bufferedScanner.hasNextLine())
    {
        line = bufferedScanner.nextLine();
    }
}
catch (Exception anException)
{
    System.out.println("Error " + anException);
}
```

6 The underlying stream is closed by sending a `close()` message to the `Scanner`, which in turn will close the underlying stream (an instance of `FileReader`). The stream must be closed from the `finally` block of the `try–catch–finally` statement. As closing the stream can throw an exception, the `finally` block has a nested `try–catch` statement. For example:

```
finally
{
    try
    {
        bufferedScanner.close();
    }
    catch (Exception anException)
    {
        System.out.println("Error " + anException);
    }
}
```

5 Persistence through serialisation

In Section 4 you implemented the `AccountsIO` class and learnt how to save the details of `Account` objects to text files in CSV format. You also learnt how to read such files in order to recreate `Account` objects from those stored details. In this way you were able to make `Account` objects persistent.

You might like to think about how you would adapt `AccountsIO` to handle collections of `CurrentAccount` objects. This would be fairly simple: all you would need to do is write additional save and load methods that take into account the additional instance variables `creditLimit` and `pin`.

But how might you adapt `AccountsIO` to handle *mixed* collections of `Account` and `CurrentAccount` objects? This is more complex. For the `saveAccounts()` method you would have to explicitly detect instances of `CurrentAccount`.

One way of doing this would be with an `if` statement whose condition used the `instanceof` operator to detect instances of `CurrentAccount`. The statement block of the `if` statement would then ensure that the extra information associated with instances of `CurrentAccount` objects (`creditLimit` and `pinNo`) was written to file. For example:

`instanceof` is an operator that tests whether the run-time class of its first operand is assignment compatible with the class name given as the second operand.

```
for (Account eachAccount : accountCollection)
{
    bufferedFileWriter.write(eachAccount.getHolder() + ","
                             + eachAccount.getNumber() +","
                             + eachAccount.getBalance());
    if (eachAccount instanceof CurrentAccount)
    {
        bufferedFileWriter.write("," + ((CurrentAccount) eachAccount).getCreditLimit()
                                 + "," + ((CurrentAccount) eachAccount).getPin());
    }
    bufferedFileWriter.newLine();
}
```

For the `loadAccounts()` method you would need another strategy. After getting the first three tokens you could use an `if` statement and a `hasNext()` message to the `Scanner` to detect whether there were any more tokens in the current line. If there were, you would know that you were dealing with a description of a current account; otherwise, you would know that it was an ordinary account. For example:

```
accountHolder = lineScanner.next();
accountNumber = lineScanner.next();
accountBalance = lineScanner.nextDouble();
if (lineScanner.hasNext()) // it is a CurrentAccount
{
    creditLimit = lineScanner.nextDouble();
    pin = lineScanner.next();
    this.accountSet.add (new CurrentAccount (holder, number, balance, creditLimit, pin));
}
else // just add an ordinary Account
{
    this.accountSet.add(new Account (holder, number, balance));
}
```

While these solutions are tractable when `Account` has only a single subclass, it becomes more difficult once we have a number of subclasses, e.g. `SaverAccount`, `PremierAccount` etc. And what if we wanted to make `Frog` and `Marionette` objects persistent? We would need to write other specialised classes to write and read them to and from file.

However, Java provides a mechanism whereby any object can be turned into a sequence of bytes (rather than characters), which can be saved to a file – and this file can then be used to reconstruct the original object. This process is known as object **serialisation**.

Important points to note about **serialisation** are:

1 For an object to be serialisable, its class (or one of its superclasses) must implement the `Serializable` interface.

2 Serialisation produces bytes, not characters. So when a serialised object is written to or read from a file, we have to use classes which read and write bytes rather than characters. So serialisation uses classes based on `InputStream` and `OutputStream` rather than the `Reader` and `Writer` classes.

In *Unit 9*, Activity 6, you discovered that when you compiled the source file `Welcome.java` the file `Welcome.class` was created. `Welcome.class` is known as the class file and contains the bytecode defining the class.

3 Retrieving a serialised object requires access to the file containing the compiled class.

5.1 The `Serializable` interface

The `Serializable` interface is a bit unusual. If you look at its documentation in `java.io`, you will find that it contains no methods, and the interface only serves to mark the implementing class as serialisable. Many of the standard Java classes implement the `Serializable` interface. If a class implements `Serializable` then any subclasses will also inherit the behaviour.

For example, if we want to make instances of the `Account` class serialisable then the `Account` class needs to implement the `Serializable` interface. This requires just two simple changes to the `Account` class.

1 Include the following import statement as the first line in the file containing the `Account` class:

```
import java.io.Serializable;
```

Alternatively, just use

```
import java.io.*;
```

(to import every class in the `java.io` library).

2 Modify the first line of the definition of the `Account` class from

```
public class Account
```

to:

```
public class Account implements Serializable
```

That is all there is to making instances of a class serialisable!

5.2 Reading and writing serialised objects

As noted above, serialisation involves bytes rather than characters, and so we use an instance of the `ObjectInputStream` class to read serialised objects from a file, and an instance of the `ObjectOutputStream` class to write serialised objects to a file. Reading from and writing to instances of `ObjectInputStream` and `ObjectOutputStream` is very similar to using `Reader` and `Writer` classes and involves wrapping other `InputStream` and `OutputStream` classes and catching the checked exceptions.

Writing a serialised object to a file

The overall structure of the code for writing a serialised object to a file is very similar to that for writing characters to a text file:

```
String pathname = OUFileChooser.getFilename();
File aFile = new File(pathname);
ObjectOutputStream bufferedOutputStream = null;
try
{
    bufferedOutputStream = new ObjectOutputStream
            (new BufferedOutputStream(new FileOutputStream(aFile)));
    // writing a serialised object to the stream goes here
}
```

The `catch` and `finally` blocks are then the same as you have seen many times before.

Note how an instance of the `ObjectOutputStream` class wraps an instance of `BufferedOutputStream` which in turn wraps an instance of `FileOutputStream`. If we were not bothered about buffering we would just create our instance of `ObjectOutputStream` as follows:

```
new ObjectOutputStream(new FileOutputStream(aFile));
```

Writing an object to the stream is accomplished by sending a `writeObject()` message to the `ObjectOutputStream`. The `writeObject()` method has the following method heading:

```
public final void writeObject(Object obj)
```

Hence `writeObject()` can be used to write an object of any class that implements the `Serializable` interface, as all objects are type compatible with `Object`. As the collections in the Collections Framework all implement the `Serializable` interface, the `writeObject()` method can be used to write whole collections of objects that also implement the `Serializable` interface, all in a single write.

When creating the `ObjectOutputStream` object there are two checked exceptions that could be thrown:

- The `FileOutputStream` constructor will throw a `FileNotFoundException` if either `aFile` cannot be created or it exists but cannot be opened.
- The `ObjectOutputStream` constructor will throw an `IOException` if an error occurs when the stream is created.

As with a `BufferedWriter`, the constructor of a `BufferedOutputStream` does not throw any exceptions.

If we look at the documentation for the `writeObject(Object obj)` method we can see which exceptions it may throw:

Throws:

`InvalidClassException` – Something is wrong with a class used by serialization.

`NotSerializableException` – Some object to be serialized does not implement the `java.io.Serializable` interface.

`IOException` – Any exception thrown by the underlying `OutputStream`.

Reading a serialised object from a file

The code for reading a serialised object from a file is, apart from using `InputStream` rather than `OutputStream` classes, very similar to that for writing a serialised object:

```
String pathname = OUFileChooser.getFilename();
File aFile = new File(pathname);
ObjectInputStream bufferedInputStream = null;
Object anObject = null;
try
{
    bufferedInputStream = new ObjectInputStream
            (new BufferedInputStream(new FileInputStream(aFile)));
    anObject = bufferedInputStream.readObject();
}
```

Note how an instance of the `ObjectInputStream` class wraps an instance of `BufferedInputStream` which in turn wraps an instance of `FileInputStream`. If we were not concerned about buffering we would just create our instance of `ObjectInputStream` as follows:

```
new ObjectInputStream(new FileInputStream(aFile));
```

Reading an object from the stream is accomplished by sending a `readObject()` message to the `ObjectInputStream`. The `readObject()` method has the following method heading:

```
public final Object readObject() throws IOException, ClassNotFoundException
```

Notice that `readObject()` returns an instance of `Object`. It knows nothing about the *actual* class of the object it has read (it's just bytes); so to do anything useful with the recovered object, you need to know what class of object to expect and then cast the returned object reference into the expected type. The class of any object can be determined by sending it a `getClass()` message.

When creating the `ObjectInputStream` object there are two checked exceptions that could be thrown:

- The `FileInputStream` constructor will throw a `FileNotFoundException` if either `aFile` cannot be created or it exists but cannot be opened.
- The `ObjectInputStream` constructor will throw an `IOException` if an error occurs when the stream is created.

As with a BufferedReader, the constructor of a BufferedInputStream does not throw any exceptions.

The method readObject() throws two checked exceptions, an IOException and, if the compiled class of the serialised object cannot be found, a ClassNotFoundException.

ACTIVITY 17

Launch BlueJ and open Unit12_Project_8. This project includes a class called ObjectIO and a version of the Account class which implements the Serializable interface. Open the class ObjectIO in the editor and complete the static methods saveObject() and retrieveObject(), so that their behaviour matches their method headings. Once you have got the ObjectIO class to compile, open the OUWorkspace.

1 Choose Open from the OUWorkspace's File menu and select the file Activity17.txt. This will load into the Code Pane the code needed to create a mixed set of Account and CurrentAccount objects referenced by the variable initialSet. Select and execute the code. Then inspect initialSet.

2 Test your saveObject() method by invoking it on the class ObjectIO with initialSet as its argument. When prompted for a file name enter accounts.dat. Note that we have given the file name a .dat extension, to indicate that it contains raw data in the form of bytes rather than text.

   Open the file accounts.dat using Notepad, and look at its contents. Close the file, and if prompted *do not* save any changes

3 Next test your retrieveObject() method by writing and executing the following statement:

```
recoveredSet = ObjectIO.retrieveObject();
```

   Finally inspect recoveredSet, does the set it references contain the objects you expected?

DISCUSSION OF ACTIVITY 17

Your saveObject() method should be similar to this:

```
public static void saveObject(Object anObject)
{
    String pathname = OUFileChooser.getFilename();
    File aFile = new File(pathname);
    ObjectOutputStream outStream = null;
    try
    {
        outStream = new ObjectOutputStream
                (new BufferedOutputStream(new FileOutputStream(aFile)));
        outStream.writeObject(anObject);
    }
    catch (Exception anException)
    {
        System.out.println("Error: " + anException);
    }
```

```
        finally
        {
            try
            {
                outStream.close();
            }
            catch (Exception anException)
            {
                System.out.println("Error: " + anException);
            }
        }
    }
```

Your `retrieveObject()` method should be similar to this:

```
    public static Object retrieveObject()
    {
        String pathname = OUFileChooser.getFilename();
        File aFile = new File(pathname);
        ObjectInputStream inStream = null;
        Object anObject = null;
        try
        {
            inStream = new ObjectInputStream
                    (new BufferedInputStream(new FileInputStream(aFile)));
            anObject = inStream.readObject();
        }
        catch (Exception anException)
        {
            System.out.println("Error: " + anException);
        }
        finally
        {
            try
            {
                inStream.close();
            }
            catch (Exception anException)
            {
                System.out.println("Error: " + anException);
            }
        }
        return anObject;
    }
```

When you execute the statements loaded from file `Activity17.txt`:

1. The variable `initialSet` references a set containing three `Account` objects and one `CurrentAccount` object.
2. The file `accounts.dat` contains bytecode, which is not designed to be read by humans! When you open it in Notepad you will be able to pick out some words, but there are many unintelligible characters.
3. The variable `recoveredSet` references a set containing clones of the objects in `initialSet`. The `CurrentAccount` object in `initialSet` has been successfully retrieved as a `CurrentAccount` in `recoveredSet`.

If you had any problems with Activity 17, the completed class `ObjectIO` *has been added to Unit12_Project_9.*

5.3 The limitations of serialisation

Serialisation is a much easier way of achieving persistence than explicitly writing the details of an object as characters to a text file. However, in order to recover a serialised object, the Java virtual machine must know where to find the compiled class of the object (i.e. its `.class` file). For this reason, when a method in a class (or indeed the OUWorkspace) attempts to recover a serialised object from a file, the compiled `.class` file of the serialised object (and the `.class` files for the objects referenced by its instance variables) must be 'in scope'. That is the files should be in the same project or brought into scope in some other way – perhaps through an `import` statement. If the compiled class(es) cannot be found, the `readObject()` method of `ObjectInputStream` will throw a `ClassNotFoundException`. In the case of a serialised collection object, the Java virtual machine must know where to find the compiled class of the collection class *and* the class of its element type *and* the classes of the element type's instance variables.

So if you want to save an object using serialisation to a file and then perhaps email that file to someone else, you must make sure that the recipient also has access to the classes on which that object is based.

This can be contrasted with saving the details of an object (values of the instance variables) as characters in a text file and sending it to someone. The file can be read by a human, and provided you tell them how the information is set out in the file (for example, account holder followed by account number followed by balance), they will be able to make use of that information (if only by reading) without having the original `Account` class.

6 Streams and networks

Streams have an integral role in network programming and distributed computing. These issues are covered in detail in other courses, but in this short section we shall briefly outline how streams are used in this context.

The Java network classes (found in the package `java.net`) are designed to simplify development of programs to run across networks, particularly the Internet. Sending and receiving information across a network is achieved using streams, so once a connection has been made between two machines, data can be transferred between the machines in exactly the same way as reading and writing to and from files.

If we want to transfer information between two computers over a network connection, one computer will have the role of the **server** and the other the role of the **client**. The connection is achieved using sockets. A **socket** represents one end of a connection between two machines, and a connection is made between a socket on the server and the socket on the client.

A computer may have many server and/or client applications running at any one time, but will usually only have one physical port for sending and receiving data across a network (an Ethernet port). Therefore there is a need to ensure that data is directed to the appropriate applications. This is achieved by the use of **port numbers** which identify virtual (software) ports. Each server application is associated with a particular virtual port (identified by its port number). Data is transferred across networks in chunks of data called packets. Each packet contains the IP (internet protocol) address of the destination computer and the port number of the intended application. When a packet arrives at the computer (identified by its IP address), the operating system reads the port number and directs the packet to the intended application.

The server

When a server is started, it waits for a connection from a client. A server waits for a connection by 'listening' to a specific port number which represents the service, or program, which the server is willing to offer to a client.

Note that in all the code samples given in this section `try-catch` statements have been omitted for simplicity.

This waiting and listening for a connection is implemented through an instance of the `ServerSocket` class as illustrated below:

```
Socket clientSocket = null;
ServerSocket listener = new ServerSocket(9999);
clientSocket = listener.accept();
```

The `accept()` method begins listening to the specified port (`9999` in this example) and execution of the method does not end until a connection is requested from a client on that port. At that point, `accept()` accepts the connection, and creates and returns an instance of `Socket` (which represents the client end of the connection). In the above code this instance is assigned to the variable `clientSocket`.

To allow the server to read from, and write to, the client, a socket has both input and output streams associated with it, which are accessed by the messages `getInputStream()` and `getOutputStream()`. Theses streams, just like all the others encountered in this unit, can be wrapped by instances of other stream classes for ease of use.

For example, to write to the client we could write:

```
PrintWriter outStream = new PrintWriter(clientSocket.getOutputStream());
outStream.println("Hello Client");
```

Similarly to read from the client we could use:

```
BufferedReader inStream = null;
String stringFromClient;
inStream = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
stringFromClient = inStream.readLine();
```

Note `try-catch` statements have been omitted from the code above for simplicity.

The client

In order to contact the server the client needs to know two things:

- The IP address of the server. Every machine which connects to the Internet has to have an IP address, such as `123.235.189.12`.
- The port number of the service which the client wants to use.

Using this information, a client can then create a socket to the server as follows:

```
Socket serverSocket = new Socket(123.235.189.12, 9999);
```

The input and output streams of the server socket can then be obtained and wrapped to allow the client to read from, and write to, the server socket.

```
PrintWriter outStream = new PrintWriter(serverSocket.getOutputStream());
outStream.println("Hello Server");
```

Similarly to read from the server

```
BufferedReader inStream = null;
String stringFromServer;
inStream = new BufferedReader(new InputStreamReader(serverSocket.getInputStream()));
stringFromServer = inStream.readLine();
```

Again, `try-catch` statements have been omitted from the code above for simplicity.

This communication between client and server is summarised in Figure 11.

You can see that once the input and output streams of the sockets have been wrapped appropriately, reading and writing data across a network uses similar classes and message-sends as reading and writing data to and from files.

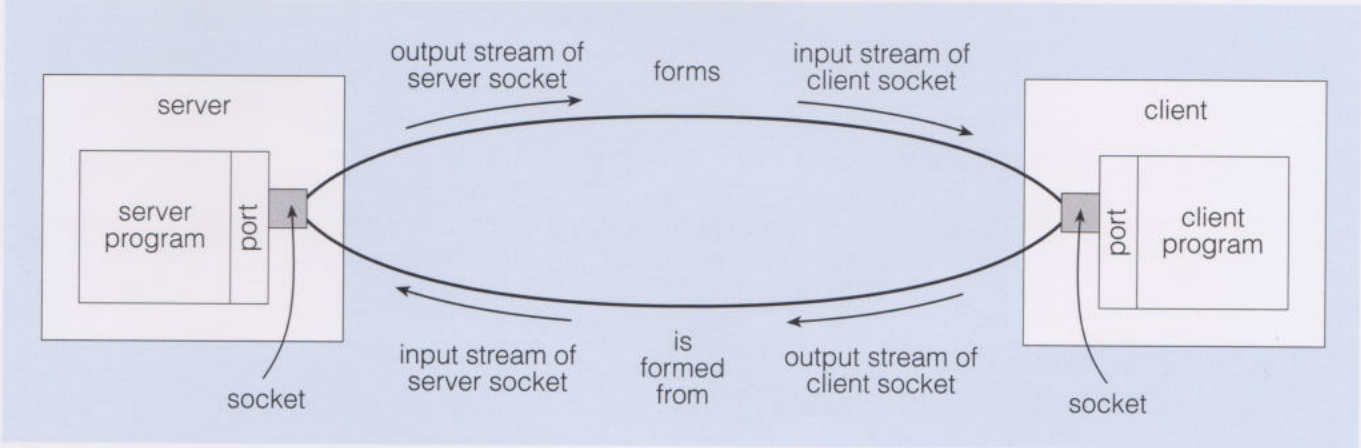


Figure 11 Client-server communication

7 Summary

After studying this unit you should understand the following ideas.

- Files may be used to make data persistent.
- Streams may be used to transfer data from a source to a sink, where the source may be a program, a file, a keyboard or a network connection, and the sink may be a program, a file, a computer monitor or a network connection.
- The `Reader` and `Writer` classes are used to read and write character data (to and from files), while the `InputStream` and `OutputStream` classes are used to read and write binary data (to and from files).
- Exceptions may occur when using streams, and these are handled using `try–catch–finally` statements.
- The details of objects can be written to file as character data, which can then be later read from file to re-create those objects.
- Serialisation enables objects to be saved as binary data in files, with advantages in terms of efficiency but some disadvantages in terms of transparency and portability.

LEARNING OUTCOMES

After studying this unit you should be able to:

- appreciate the range of classes in the `java.io` library;
- understand how stream classes are wrapped;
- understand the difference between checked and unchecked exceptions;
- handle exceptions appropriately;
- write methods which allow character-based information to be read from or written to an external file;
- write methods to save the details of objects as character data in a file;
- write methods to read the details of objects saved as character data in a file and, from that data, re-create those objects;
- understand how objects may be serialised;
- write methods which allow objects to be serialised and saved to file as binary data;
- write methods which read serialised objects from files.

Glossary

buffer An area used for temporary storage as data is transferred between a data source and a data sink.

checked exception An exception which has to be caught by a `try–catch` statement. Checked exceptions relate to problems that can be foreseen (e.g. trying to write a file larger than the available disk space) but cannot necessarily be detected before they occur.

`File` An instance of this class contains the pathname to a file or folder in a system-independent format. The file specified need not exist; instead the pathname may point to a *potential* new file or folder.

`InputStream` The base class of a hierarchy of classes including `FileInputStream`, `BufferedInputStream` and `ObjectInputStream`, which are used to read 8-bit byte streams. `InputStream` classes are used to read binary data.

`OutputStream` The base class of a hierarchy of classes including `FileOutputStream`, `BufferedOutputStream` and `ObjectOutputStream`, which are used to write 8-bit byte streams. `OutputStream` classes are used to write binary data.

persistence The ability of objects or other data to continue in existence after a program has stopped executing.

`Reader` The base class of a group of classes including `FileReader` and `BufferedReader`, which are used to read 16-bit character streams. `Reader` classes are used to read text files

`Scanner` A class used to read string tokens from a source. In M255 the source may be a `FileReader` or a `String`.

serialisation The process by which an object implementing the `Serializable` interface can be written to an `ObjectOutputStream` as a sequence of bytes.

stream An object used to connect a data source to a data sink, enabling data to be transferred from the source to the sink.

`Writer` The base class of a group of classes including `FileWriter` and `BufferedWriter`, which are used to write 16-bit character streams. `Writer` classes are used to write text files.

Index

B
buffer 24

C
client 56
comma-delimited file (CSV) 37

E
escape character 9
exception
    checked 16
    unchecked 15

F
`File` 9
finally 28

I
`InputStream` 7
`instanceof` 49

J
`java.io` 7–8

O
`OUFileChooser` 10
`OutputStream` 7

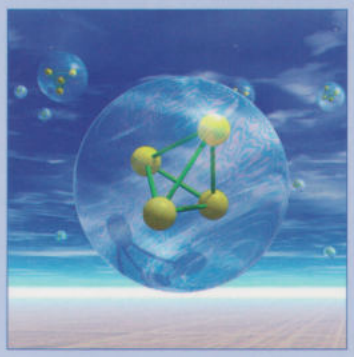
P
persistence 5, 33, 36
port numbers 56

R
`Reader` 7, 21

S
`Scanner` 38
serialisation 5, 50
server 56
socket 56
stream 6

T
token 37
`try-catch` statement 16

W
`Writer` 7, 17



M255 Unit 12
UNDERGRADUATE COMPUTING

Object-oriented programming with Java

UNIT 12

Block 4

▶ Unit 12 Streams, files and persistent objects

Unit 13 Software testing

Unit 14 Software development

M255 Unit 12
ISBN 978 0 7492 6793 3